Sistemas Operativos

FICHEROS

Daniel Melero Chaves
Datsi ETSI Informáticos de Madrid

SB	SB	Mapa Bit Agrupación	Mapa Bit i-nodo	Vector i-nodo	Ficheros

TABLAS EN MEMORIA

BCP → Tabla FD → Tabla Intermedia → Tabla cop/i-nodo

0	15
1	8
2	7
3	2

	Mod	Pos	Ref	Rw
1				
2	6	0	1	10
3				
4				

Nodo	Tipo	Fopen
1		
⋮	⋮	⋮
6	2	1

i-nodo

- 1B • Tipo de archivo (mask)
- 6B • Enlaces
- 1B • PID (prop)
- 1B • GID (prop)
- 4B • Tamaño en byte
- 3 x 4B = 12B
  - • Instante de Creación
  - • Último acceso
  - • Última modificación
- 2B Punt = 26B
  - • Puntero directo x10
  - • Puntero indirecto simple
  - • Puntero indirecto doble
  - • Puntero indirecto triple

TOTAL = 51B

TIPOS DE ENLACE

- Enlace Físico

① El fichero solo se elimina cuando se borra todos sus nombres. ② - Contador en el nodo-i del fichero. Necesario para saber cuando se puede borrar un fichero. Con cont = 0 se borra el fichero ③ - Solo se permite enlazar ficheros y subdirectorios del mismo volumen

- Enlace Simbólico

① - El fichero se elimina cuando el contador de enlaces físicos = 0. Si pertenece el enlace simbolico se provocará un error al tratar de abrirlo ② - se puede hacer con ficheros y directorios. ③ - Se puede hacer entre ficheros de diferentes volumenes

SERVICIOS FICHERO

- int open(char *name, int flags[, mode_t mode]); flags(O_ RDONLY, WRONLY, RDWR, APPEND → Se accede fin fich, CREAT); permisos
- int creat(char *name, mode_t mode);
- ssize_t read(int fd, void *buf, size_t n_bytes);
- ssize_t write(int fd, void *buf, size_t n_bytes);
- int close(int fd);
- off_t lseek(int fd, off_t offset, int whence); whence(SEEK_ SET → pos = ofset, CUR → pos = pos + ofset, END → pos = fin + ofset); desplaz
- int dup(int fd); Duplica el fd con el descriptor + ↓
- int dup2(int oldfd, int newfd); Crea un nuevo fd con el número newfd

SERVICIOS DIRECTORIO

- DIR *opendir(char *dirname);
- struct dirent *readdir(DIR *dirp);
- int closedir(DIR *dirp);
- void rewindir(DIR *dirp); Pone el puntero al inicio del dir
- int mkdir(char *name, mode_t mode);
- int rmdir(char *name);
- int link(char *existing, char *new); fichero existente / nombre nueva entrada
- int symlink(char *existing, char *new);
- int unlink(char *name);
- int chdir(char *name);
- int rename(char *old, char *new);
- char *getcwd(char *buf, size_t size);

ROOT

int mount(const char *source, const char *target, const char *filesystemtype, unsigned long mountflags, const void *data);

source → sistema a montar
target → Dir a montar
filesystemtype → sistema donde se va a montar

int umount(const char *target);

NÚMERO DE ACCESOS A MEMORIA

/user/dperez/datos.txt → lectura i-nodo → lectura dir

TAMAÑO MÁXIMO FICHERO

$$t_{max} = \left(10 + \frac{\text{Tamaño Agrup}}{\text{Tamaño direcc}} + \left(\frac{TA}{TDs}\right)^2 + \left(\frac{TA}{TDs}\right)^3\right) \cdot \text{Tamaño Agrupación}$$

(Tam Bloque)

Nº Accesos: 1, 2, 3, 4

ANEXOS

$$\text{Nº direcciones/agrupación} = \frac{\text{TamBloq} \times \text{Nº bloque/agrup}}{\text{Tam Dirs ó Ancho Dirección}}$$

$$\text{Nº máximo de agrupaciones direccionables} = 2^{\text{ancho dirección}}$$

Tamaño Agrupaciones: 512B/1KB/2KB/4KB/8KB

$$\text{Nº Agrupación} = \frac{\text{Tamaño Dispositivo (B)}}{\text{Tamaño Agrupación (B)}} = 2^x \Rightarrow x = \text{tamaño dirección}$$

$$\text{Tamaño Mapa Bits} = \frac{\text{Nº Agrupaciones}}{\text{Tamaño Agrupación}}$$

TAMAÑO METAINFORMACIÓN

(R) Tamaño Agrupación: ≥ 2 KiB
(S) Nº direcciones por agrupación
(T) Agrupaciones del fichero = TamañFichero / TamañAgrupación
(U) Direcciones en i-nodo = 10
(V) Agrupaciones para dobles indirectos
$V = (T - U - Y \cdot S)/S$
(X) Agrupaciones triples
$X = (T - U)/S - Y - V$
Tamaño Metainformación
$(V + X + Y) \cdot R + TamFich$

./incrementar archivo (M15)

```
int main(int argc, char *argv[]) {
  int fch = open(argv[1], O_RDWR|O_CREAT, 0666);
  dup2(fch,0);
  dup2(fch,1);
  close(fch);
  #define POS
  int ofs = lseek(0, POS, SEEK_SET);
  int val = 0;
  int cnt = read(0, &val, sizeof(int));
  ofs = lseek(1, -cnt, SEEK_CUR);
  val *= 5
  cnt = write(1, &val, sizeof(int));
  return 0;
}
```

*Comprobaciones de las llamadas del sscc

RECORRER DIRECTORIO

```
void recorrer_directorio (char *dire){
  DIR *dir;
  Struct dirent *entrada;
  char buffer[1024];
 dir = opendir (dire);
 while ((entrada = readdir(dir)) != NULL){
   if (strcmp (entrada->d_name, "") ==0)
     continue;
   sprintf (buffer, "%s/%s", dir, entrada->d_name);
   procesar_fichero (buffer);
 }
 closedir(dir);
}
```

1.1 Accesos a sectores: 10, 200, 75, 32, 450, 123.
Tiempo de búsqueda = 3 + 0,04x ms (x cilindros)
cilindro inicial = 300. Tiempo de búsqueda para:

- **FCFS**: orden de llegada

$t_{busqueda} = t_{busq}(300-10) + t_b(200-10) + t_b(200-75) +$
$+ t_b(75-32) + t_b(450-32) + t_b(450-123)$

- **SSF**: Brazo inmóvil

$t_{busqueda} = t_b(300-200) + t_b(200-123) + t_b(123-75) +$
$+ t_b(75-32) + t_b(32-10) + t_b(450-10)$

450 300 200 123 75 32 10

- **CSCAN**: SSF en ambos sentidos

CAMBIO DE CONTEXTO Cuando se pasa de ejutar el PA a ejecutar el PB. Se debe salvaguardar el estado en el que se encontraba ejecutando el proceso d y recuperar el estado en el que se detuvo la ejecución del PB para que pueda reanudar su ejecución en el mismo punto donde fue detenida. La intervención de so provoca dos cambios de modo en la ejecución del procesador, de usuario a privilegiado o viceversa.

1.2 Direcciones de Bloque de 32 bits

a) Tamaño del fichero con: 512B/Bloque y 4KB/Bloque

$Tamaño = \left(10 + \left(\frac{512B}{4B}\right) + \left(\frac{512B}{4B}\right)^2 + \left(\frac{512B}{4B}\right)^3\right) \cdot 512B = X$

$Tamaño = \left(10 + \left(\frac{4096B}{4B}\right) + \left(\frac{4096B}{4B}\right)^2 + \left(\frac{4096B}{4B}\right)^3\right) \cdot 4096B = Y$

b) Número de accesos para acceder a la posición 3MBytes

$N^o accesos = 10 \cdot 4096B = X$ no llega a 3MB

$N^o accesos = \underbrace{\left[10 + \left(\frac{4096}{4}\right)\right]}_{\substack{1^o \\ acceso}} \cdot 4096 = X > 3MB$ (2º acceso)

1.4 512B/sector y 150GB de capacidad. Formato Sistema UNIX. 4kB/Bloque.
Tamaño fichero = 100 kB

a) Determinar tamaños del sistema de ficheros

* Suponemos que tamaño de bloque = tamaño de agrupación

SB	SB	Mapa bits	i-nodo	Ficheros

BB (Boot Bloque) = 4kB SB (Super Bloque) = 4kB

Mapa de Bits → número i-nodos = número de agrupaciones

$\frac{150GB}{4KB/ag} = 37$ agrupaciones

Tamaño inodo (nodeinfo ≃ 2qB, 10p = 10·4kB, 1pS = 4kB, 1pD = 4kB, 1pT = 4kB)

b) Tamaño máximo fichero

$Tamaño = (10 + ()^1 + ()^2 + ()^3) \cdot 4kB = 4TB$ se pasa

$Tamaño = (10 + () + ()^2) \cdot 4kB = X < 150GB$ perfecto

M15

Teniendo 8 KiB/agrupación, tamaño del disco 0'5 TiB y tamaño medio fichero = 8 KiB. a) Capacidad de direccionamiento del i-nodo

Tamaño de las dirs a agrupación → potencia → 32 bits para direcciones

$N^o agrupaciones = 2^{39}\frac{B}{disco} \cdot \frac{agrup}{2^{13}B} = 2^{\overbrace{26}^{\text{potencia}}}\frac{agru}{disco}$

$N^o direcciones = 2^{13}\frac{B}{agru} \cdot \frac{dir}{2^2 B} = 2^{11}\frac{dir}{agrup}$ (suponemos)

$N^o agrupaciones = (10 + 2^{11} + 2^{22} + 2^{33}) \cdot 8KiB$

b) Tamaño mapa de bits

$Tamaño = 2^{26} bits \cdot \frac{B}{2^3 Bits} \cdot \frac{agrup}{2^{13}B} = 2^{10} agrupaciones$

Fichero especial orientado a carácter (un terminal) las llamadas al SSOO fallarían

PROCESOS

Daniel Melero Chaves

FORK
- Sí armado de señales
- Sí señales ignoradas
- Sí máscara de señales
- NO alarmas
- NO señales pendientes

EXEC
- NO armado de señales
- Sí señales ignoradas
- Sí máscara de señales
- Sí alarma
- Sí señales pendientes

TIPOS DE SEÑALES

SIGABRT	terminación anormal
SIGALRM	señal de fin de temporización
SIGFPE	operación aritmética errónea
SIGHUP	colgado de terminal de control
SIGILL	instrucción hardware inválida
SIGINT	señal de atención interactiva (ctrl+c)
SIGKILL	señal de terminación (kill -9) (no ignorar y no armado)
SIGPIPE	escritura en un pipe sin lectores
SIGQUIT	señal de terminación interactiva (ctrl+\)
SIGSEGV	referencia a memoria inválida
SIGTERM	señal de terminación (kill)
SIGUSR1	señal definida por la aplicación
SIGUSR2	"
SIGCHLD	indica terminación del proceso hijo
SIGCONT	continuar si está bloqueado el proceso
SIGSTOP	Señal de bloqueo (no ignorar y no armado)

MACROS (STATUS)
- WIFEXITED (status): valor + si el hijo finalizó normalmente
- WEXITSTATUS (status): valor devuelto por el proceso hijo si finalizó normalmente
- WIFSIGNALED (status): valor + si el proceso finalizó por una señal
- WTERMSIG (status): nº de señal que provocó la finalización del proceso

PIPE

fd[1] → ▭ → fd[0]

TABLA EXEC

FUNCION	Pathname	filename	arg list	argv[]	environ	envp[]
execl	X		X		X	
execlp		X	X		X	
execle	X		X			X
execv	X			X	X	
execvp		X		X	X	
execve	X			X		X
execvpe		X		X		X

pathname → Ruta Completa del ejecutable
filename → Nombre del ejecutable
arg list → argumento que quieras pasarle al exec (argumentos individuales)
argv[] → array que contiene los argumentos (pasas el puntero del array)
environ → entorno de ejecucion que quieras pasarle al exec
envp[] → array que contiene el entorno de ejecución (pasas el puntero)

SERVICIOS

pid_t wait(int *status) [Id y estado termi] [Espera hasta que un hijo termine]

pid_t waitpid(pid_t pid, int *status, int options) [Espera al proceso pid]

pid_t exit(int status) [Finaliza la ejecución de un proceso indicando el estado de terminación del mismo]

BUCLES INFINITOS

```
while (wait(&status) != pid)
    continue; → Espera a los hijos
while (wait(&status) > 0)
    continue;
```

SERVICIOS DE SEÑALES

int kill(pid_t pid, int sig); Envia al proceso pid la señal sig

int sigaction(int sig, struct sigaction *act, struct sigaction *oact);
- act.sa_handler → función de armado, SIG_IGN o SIG_DFL
- act.sa_mask → señales a bloquear durante ejecución de función de armado (tambien la propia sig)
- act.sa_flags → Opciones diversas. Por defecto se pone 0.

int pause(void) Bloquea el proceso hasta recibir una señal

unsigned int alarm(unsigned int seconds); Genera la recepción de la señal SIGALRM pasados seconds segundos

int sleep(unsigned int seconds); El proceso despierta por haber superado los seconds o por recibir una señal

SERVICIOS DE THREADS

int pthread_create(pthread_t *thread, const thread_attr_t *attr, void *(*func)(void *), void *arg)
- Crea un thread que ejecuta fun con argumento arg y atributos attr
- Los atributos permiten especificar: tamaño de la pila, prioridad, política de planificación, etc
  - → joinable: el thread no desaparece hasta que otro thread haga pthread_join de él
  - → detached: el thread desaparece al terminar
- Existen diversas llamadas para modificar los atributos

int pthread_join(pthread_t thid, void **value)
- suspende la ejecución de un thread hasta que termine el thread con identificador thid
- Devuelve el estado de terminación del thread

int pthread_exit(void *value)
- Permite a un thread finalizar su ejecución, indicando el estado de terminación del mismo

int sched_yield(void)
- El thread que la ejecuta cede el control a otro thread listo para ejecutar

int pthread_yield(void)
- Cesión voluntaria del procesador

ARMADO DE UNA SEÑAL

```
struct sigaction act;
act.sa_handler = función_trata;
act.sa_flags = 0;
sigaction (SIG_X, &act, NULL);
                  señal a tratar
sigemptyset (&act.sa_mask);
```

MASCARA A UNA SEÑAL

```
sigset_t mascara;
sigaddset (&mascara, SIG_X);
        señal a enmascarar
sigprocmask (SIG_SETMASK, &mascara, NULL);
```

PADRE LECTOR - HIJOS ESCRITORES

```
#define N 10
#define T 3
int f[2];
char buf;
int i;
void f1(int s){
   write (f[1], &buf, sizeof(buf));
}
int main(void){
 pid_t p[N];
 if (pipe(f) <0){
  perror("pipe\n");
  return 1;
 }
 struct sigaction a1;
 sigemptyset (&a1.sa_mask);
 a1.sa_handler = f1;
 a1.sa_flags =0;
 sigaction (SIGUSR1, &a1, NULL);
 for (i=0; i<N; i++){
   if ((p[i] = fork()) ==-1 ){
      perror("fork");
      return 1;
   }else if (p[i] == 0){
     close (f[0]);
     buf = '0' + i;
     pause();
     return 0;
   }
 }
 close (f[1]);
 sleep(T);
 for (i=0; i<N; i++){
    kill (p[i], SIGUSR1);
 }
 for (i=0; i<N; i++){
   if (read (f[0], &buff, 1) <0) { perror(
 }
 return 0;
```

THREDS

```
pthread_t Id_B=0, Id_D=0;
int a, b, c, d, e;
void *B (void *p){
   b = tarea_B(0);
   pthread_exit(NULL);
}
void *D (void *p){
   d = tarea_D(a);
   pthread_exit(NULL);
}
int main (void){
 pthread_attr_t attr;
 pthread_attr_init (&attr);
 pthread_attr_setdetachstate
     (&attr, PTHREAD_CREATE_JOINABLE);
 pthread_create (&Id_B, &attr, &B, NULL);
 a = tarea_A(0);
 pthread_create (&Id_D, &attr, &D, NULL);
 c = tarea_C(a);
 pthread_join (Id_B, NULL);
 e = tarea_E(b+c);
 pthread_join (Id_D, NULL);
 pthread_attr_destroy (&attr);
 printf("%i %i %i %i %i\n", a, b, c, d, e);
}
```

TURNOS PADRE → HIJO → NIETO

```
pid_t p1;
void procesa_peticion(char b);
void f1(int IDsenal){
  char b;
  read (0, &b, 1);
  kill (p1, SIGUSR1);
  procesa_peticion(b);
}
int main(void){
 pid_t p;
 int i;
 struct sigaction act;
 act.sa_handler = f1;
 act.sa_flags = 0;
 sigaction (SIGUSR1, &act, NULL);
 p1 = getpid();
 for (i=0; i<N; i++){
   switch (p = fork()){
     case -1:
       perror("fork()");
       exit(1);
     case 0:
       while(1)
         pause();
     default:
       p1 = p;
   }
 }
 kill (p1, SIGUSR1);
 while(1)
  pause();
}
```

N15

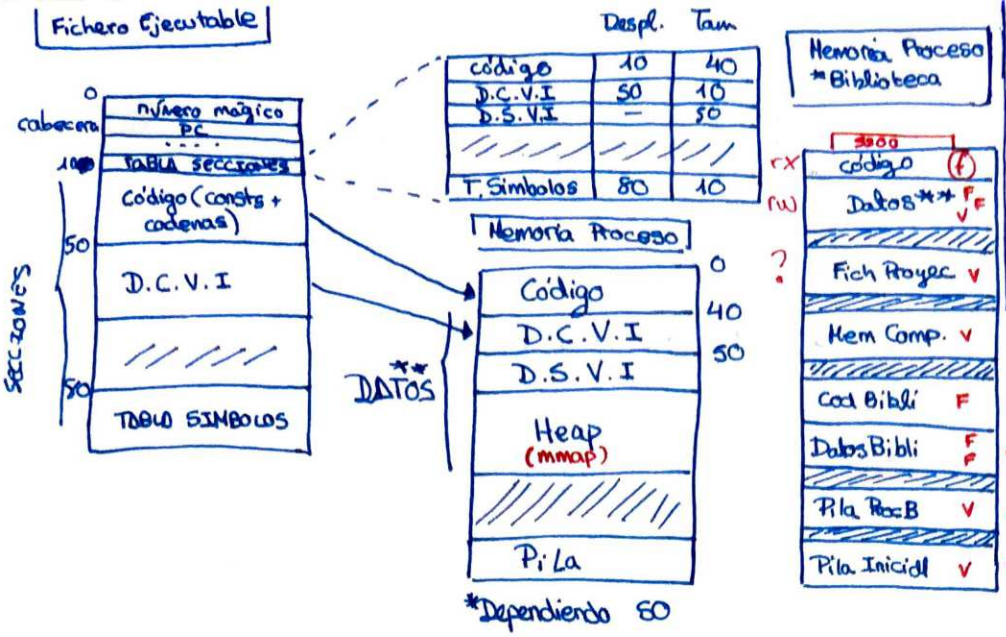
TA) act.sa_handler = tratar_alarma TB) act.sa_handler = SIG_DFL;

TB ← P → TA

① Alarma vence antes TA y TB → El proceso padre muere ② TA, TB, vence alarma → TA protege de la alarma, TB cambia y desprotege de la alarma, vence la alarma el proceso padre muere ③ TB, TA, vence alarma → TB desprotege de la alarma, TA cambia y protege de la alarma, vence la alarma se hace tratar_alarma y se sigue con la ejecución ④ TA, vence alarma, TB → TA protege de la alarma, vence la alarma se hace tratar_alarma y se sigue con la ejecución y se crea a TB ⑤ TB, vence alarma, TA → TB desprotege de la alarma la alarma salta y el proceso muere, no se crea TA

Daniel
MELERO
CHAVES

Fichero Ejecutable

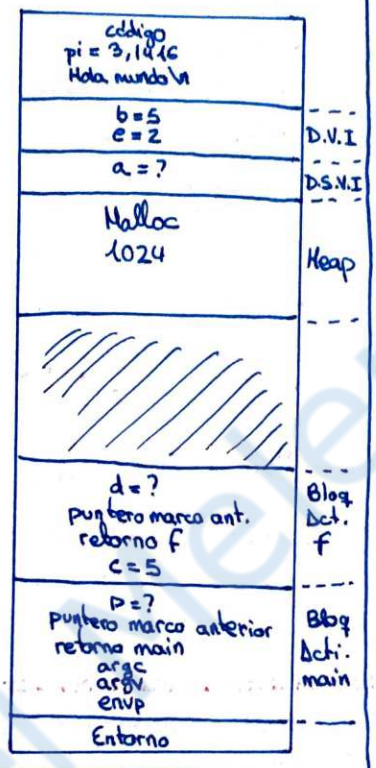


* Datos Biblioteca no tiene Heap

código → comp / ejecutable
Datos → priv / ejecutable
Fich Proyec → priv / fichero
Mem. Comp → comp / fichero
Cod Bibli → comp / bibli
Datos Bibli → ~~comp~~ priv / bibli
Pila ProcB → priv
Pila Main → priv

Ejemplo

```
int a;
int b = 5;
const float pi = 3.1416;
void f(int c){
  int d;
  static int e = 2;
  d = 3;
  b = d + 5;
  ......
  return;
}
int main(int argc, char *argv[]){
  char *p;
  p = malloc(1024);
  f(b);
  ......
  free(p);
  printf("Hola Mundo\n");
  return 0;
}
```



Anotaciones

Boot SIZE B	Sup. Bloq SIZE B	MAPA *	Nodos-i **	FICHEROS	SWAP

** i-nodo por bloque (usamos la mitad)

$$\text{Tamaño} = \frac{\text{nº inodos}}{2} \cdot \underbrace{\text{puntero y mierdas}}_{(20)} \cdot 4\left(\frac{\text{KiB}}{\text{Bloq}}\right)$$

*
$$\text{Mapa} = \frac{\text{nº bloques} + \text{nº i-nodos}/2}{2 \cdot \left(\frac{B}{\text{Bloq}}\right) 4}$$

* mandato size (B)
text = código
data = DCVI
bss = DSVI
⇒ dec y Hex = tamaño total

* Error mmap
if (m = MAP_FAILED) perror ("mmap");

* Curiosidades 1
→ Read: Fichero Proyectado (MAP_SHARED)
→ Texto si read → No (constantes ni cadenas texto)
→ Si no están las páginas en memoria principal (ejecutado recientemente)

Tabla de Páginas (sin segmentos)

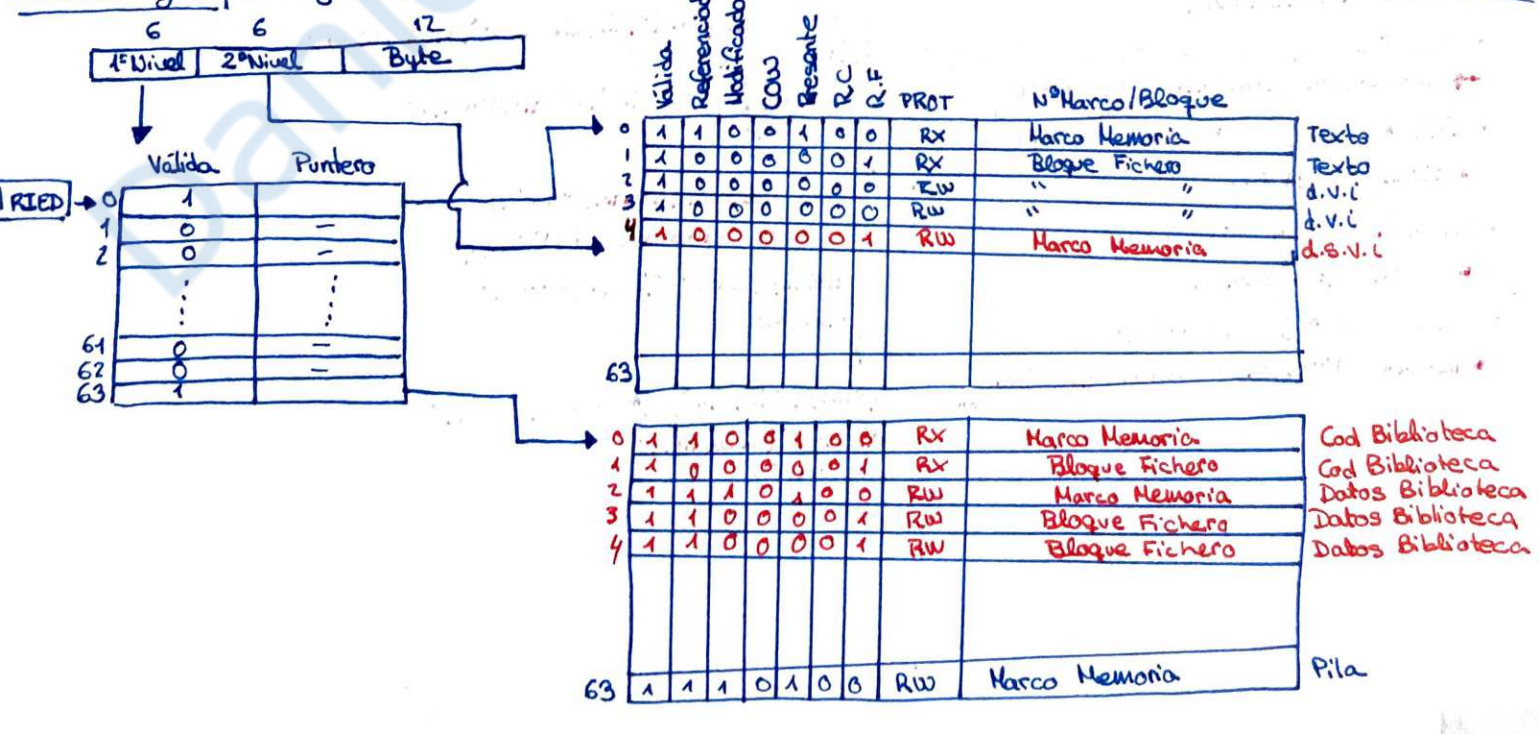
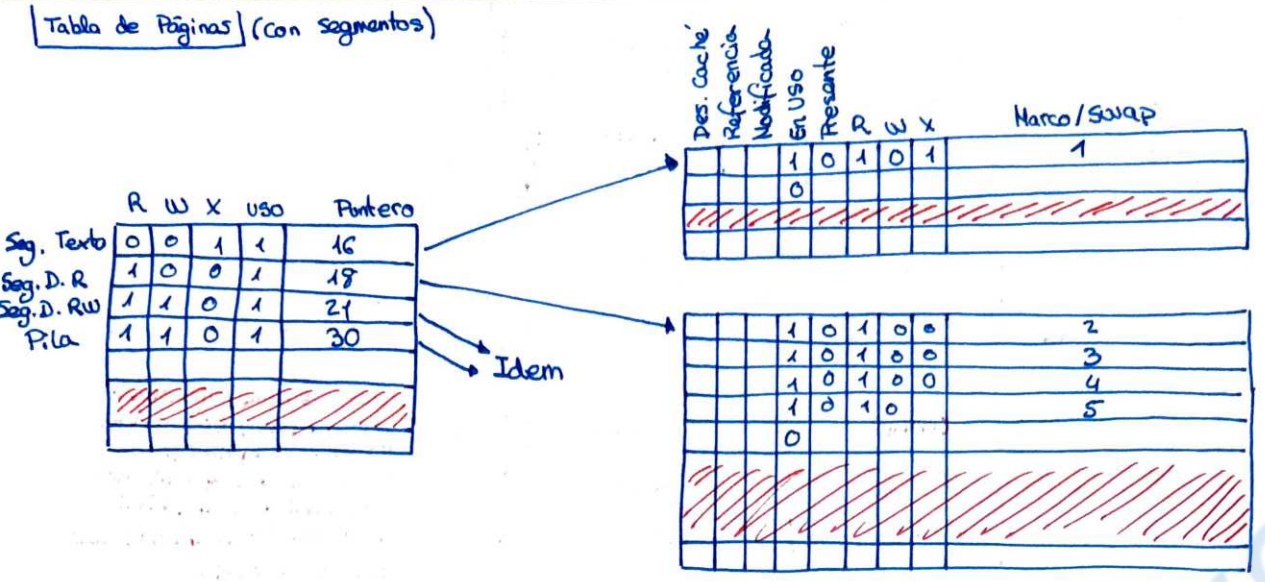


Tabla de Páginas (con segmentos)



SERVICIOS MEMORIA

```
void * mmap (void * addr, size_t len, int prot, int flags,
             int fildes, off_t off); // proyecta un fich
```

- → addr = dir de comienzo (NULL)
- → len = nº bytes a proyectar (páginas enteras)
- → prot = tipo de acceso
  - PROT_READ/WRITE/EXEC o NONE
- → flags = manejo de los datos
  - MAP_SHARED/PRIVATE/ANONYNOUS....
- → fildes = fd a proyectar
- → off = desplazamiento dentro del fichero

```
void munmap (void *addr, size_t len);
```

- → addr = dirección de comienzo
- → len = nº bytes a desproyectar

Anotaciones

* Direcciones en hexadecimal

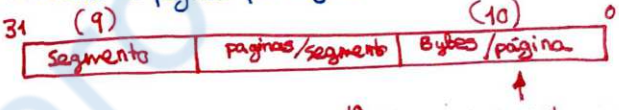
2 páginas → 1 página = 1 KiB

2 KiB = 2048 B

2048 | 16 → 0, 128 | 16 → 0, 8, ⇒ Dirección 2 páginas

800 → 00...000 − 00...7FF

* Número de páginas por segmento



1 KiB/página = 1024 B/pag = 2^{10} B/págin = 10 bits

32 − 9 − 10 = 13 bits/páginas

2^{13} páginas = 8192 páginas

H15

```
int *p;
int main(){
p=mmap(...);
}
```

¿Comparten los threads el mmap?

Sí, tienen distintas pilas en memoria pero comparten la mmap (P es global)

SERVICIO BIBLIOTECA

- void *dlopen (const char *filename, int flag);
  - → Devuelve un manejador (var ok) ó (!var error)
- Void *dlsym (void *handle, char *symbol);
  - → NULL o !NULL
- int dlclose (void *handle);
  - → 0 OK ≠0 ERROR
- const char *dlerror (void);
  - → Devuelve NULL OK ó Cadena con (Error)

MONTAJE EXPLÍCITO DE BIBLIOTECA DINÁMICA

```
int main (int argc, char **argv){
  void *handle;
  double (*cosine)(double); //Puntero a función
  char *error;
  handle = dlopen ("/lib/libm.so", RTLD_LAZY);
  if(!handle){
    fprintf (stderr, "%s\n", dlerror());
    exit(1);
  }
  cosine = dlsym (handle, "cos"); // Valor del puntero
  if ((error = dlerror()) != NULL){
    fprintf (stderr, "%s\n", error);
    exit(1);
  }
 printf ("%f\n", (*cosine)(2.0)); //Obtenemos la operación
 dlclose (handle);
 return 0;
}
```

Semáforos

- sem_t nombre
- sem_init(nombre, compartido?, valor);
- sem_wait(nombre);
- sem_post(nombre);
- sem_destroy(nombre);
- sem_t *sem_open(char *name, int flag, mode_t mode, int valor_inicial);
- int sem_close(sem_t *nombre);
- int sem_unlink(char *nombre);

Daniel
Melero
Chaves

Mutex y Condiciones

* mutex/cond

- pthread_mutex_t nombre;
- pthread_cond_t nombre;
- pthread_mutex_lock(&nombre, NULL);
- pthread_mutex_unlock(&nombre, NULL);
- pthread_*_init(&nombre, NULL);
- pthread_*_destroy(&nombre);
- pthread_cond_wait(&nCond, &nMutex);
- pthread_cond_signal(&nCond);
- pthread_cond_broadCast(&nCond);

mutex_unlock(m) {
if(alguien esperando)
//entregar llave
else
//Quedarmela

condition_signal(c) {
if(alguien esperando)
//avisar que siga
else
//Se pierde el signal

Cerrojos

- struct flock {
  short l_type; // F_RDLCK (Compartido), F_WRLCK (Exclu), F_UNLCK (Eliminar)
  short l_where; // SEEK_ {SET, CUR, END}
  off_t l_start; // Desfase
  off_t l_len; // Tamaño Cerrojo
  pid_t l_pid; // F_GETLK primer pid con lock
  }
- int fcntl(fd, cmd, [nombre_flock]);
  → cmd: F_GETLK → ¿Hay cerrojo?
  F_SETLK → No Bloq, asinc
  F_SETLKW → Bloq, sinc

Transacciones

* Idea conceptual del Cliente-Servidor
⇒ Un proceso no puede avanzar sino obtiene respuesta de otro

- struct flock fl; fl.l_where = SEEK_SET;
  fl.l_start = 0, fl.l_len = 0; (cierra todo fich) fl.l_pid = getpid();

```
fl.l_type = F_RDLCK; // Compartido
fcntl(fd, F_SETLKW, &fl); // Ponemos cerrojo
lseek(fd, 0, sizeof(int)); // Posición en el fichero
  //Operaciones intermedias
fl.l_type = F_UNLCK; // Cerramos cerrojo
fcntl(fd, F_SETLK, &fl);
```

Mecanismos

	Threads	PP_Parientes	PP_Locales	PP_Remotos
Sincronización	mutex + cond; Semaforo sin nombre		Semaforo con nombre; Cerrojo sobre fichero	Transacciones más cerrojos
Avisar		Señales		
Almac.	Imagen de memoria	Memoria Compartida	Fichero Proyectado	
		Fichero		
Comunicar		PIPE	FIFO; SOCKET UNIX	SOCKET INET

Comunicación Local

PIPE

Creación: int pipe(int pp[2]);
Lectura: int read(int pp[0], datos, n);
Escritura: int write(int pp[1], datos, n);

FIFO (Fichero)

Creación: int mkfifo(char *name, mode_t mode);
Eliminar: int unlink(char *name);
Abrir: int open(char *name, int flags);
Lectura: int read(fd_in, datos, n);
Escribir: int write(fd_out, datos, n);

Comunicación Remota

- int socket(dominio, tipo, protocolo);
  → Dominio (int)
  AF_UNIX → Intra-Máquina
  AF_INET → Entre-Máquinas
  → Tipo (int)

SOCK_STREAM	SOCK_DGRAM
• Flujo Datos	• Mensajes
• CON conexión	• SIN conexión
• Fiable	• No fiable
(Llamada teléfono)	(correo postal)

  → Protocolo (int)
  En AF_INET
  → IPPROTO_TCP: stream
  → IPPROTO_UDP: datagrama

- Socket Dir Local: int bind(int sd, struct sockaddr *dir, int tam);
- socket remoto Dir (cliente): int connect(int sd, struct sockaddr *dir, int tam);
- Preparar para aceptar conexión (servidor): int listen(int sd, int backlog);
- Aceptación de una conexión (servidor): int accept(int sd, struct sockaddr *dir, int tam);
- Transmisión: int send(int sd, char *mem, int tam, int flags); int recv(int sd, char *mem, int tam, int flags);

Estructuras

- Cliente-Servidor: Ventajas: Control centralizado que facilita la tarea de poner al día los datos y otros recursos. Escalabilidad, se puede aumentar la capacidad de clientes y servidores por separado.
  Desventajas: Congestión de tráfico
- Peer-to-Peer: Ventajas: Estabilidad. Robustez, ficheros distribuidos en varios equipos, si falla 1 puede suplantarle otro
  Desventajas: Sincronización de datos, no centralizado.

```
void Productor (void){
  int i, dato;
  for (i=0; i<TDatos; i++){
    //Producimos
    pthread_mutex_lock(&mutex);
    while (nDatos == MAXBUF)
      pthread_cond_wait(&lleno, &mutex);
    buffer[i% MAXBUFF] = dato;
    n_dato++;
    pthread_cond_signal(&novacio, &mutex);
    pthread_mutex_unlock(&mutex);
  }
}

void Consumidor (void){
  int i, dato;
  for(i=0; i<TDatos; i++){
    pthread_mutex_lock(&mutex);
    while (n_datos==0)
      pthread_cond_wait(&novacio, &mutex);
    dato = buffer[i % MAXBUFF];
    n_dato--;
    pthread_cond_signal(&lleno);
    pthread_mutex_unlock(&mutex);
  }
}

int main (void){
  pthread_mutex_t mutex
  pthread_cond_t cond
  pthread_t th1
  ---------- ----
  pthread_mutex_init(&mutex, NULL)
  pthread_cond_init(&cond, NULL)
  ---------- ----
  pthread_create(&th1, N, Programa, N);
  pthread_join(th1, N)
  ---------- ----
  pthread_mutex_destroy(&mutex)
  pthread_cond_destroy(&cond)
  return 0;
}
```

Lector - Escritor → Mutex

```
void Lector (void){
  pthread_mutex_lock(mutex);
  while (escribiendo != 0)
    pthread_cond_wait(aleer, mutex);
  leyendo++;
  pthread_mutex_unlock(mutex)
  //Lecturas del recurso
  pthread_mutex_lock(mutex)
  leyendo--;
  if (leyendo ==0)
    pthread_cond_signal(aescribir)
  pthread_mutex_unlock(mutex)
}

void Escritor (void){
  pthread_mutex_lock(mutex);
  while (leyendo!=0 || escribiendo!=0)
    pthread_cond_wait(aescribir)
  escribiendo++;
  pthread_mutex_unlock(mutex)
  //Escritura del recurso
  pthread_mutex_lock(mutex)
  escribiendo--
  pthread_cond_signal(aescribir)
  pthread_cond_broadcast(aleer)
  pthread_mutex_unlock(mutex)
}
```

Tamaño Fichero

- struct stat st;
  fstat(fd, buff, &st);
  st.st_size
- lseek(fd, 0, SEEK_END);

NOTAS

mmap(0, tam, PROT_*, MAP_SHARED, fd, 0)
n = atoi(argv[2]);

Lector - Escritor → Cerrojo

```
void Lector (void){
  int fd, val, i;
  struck flock fl;
  fl.l_whence = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;
  fl.l_pid = getpid();
  fd = open("BD", O_RDONLY);
  for(i=0; i<10; i++){
    fl.l_type = F_RDLCK;
    fcntl(fd, F_SETLKW, &fl);
    lseek(fd, 0, SEEK_SET);
    read(fd, &val, sizeof(int));
    fl.l_type = F_UNLCK;
    fcntl(fd, F_SETLK, &fl);
  }
  return 0;
}

void Escritor (void){
  int fd, val, i;
  struck flock fl;
  fl.l_where = SEEK_SET;
  fl.l_start = 0;
  fl.l_len = 0;
  fl.l_pid = getpid();
  fd = open("BD", O_RDWR);
  for(i=0; i<10; i++){
    fl.l_type = F_WRLCK;
    fcntl(fd, F_SETLKW, &fl);
    lseek(fd, 0, SEEK_SET);
    read(fd, &val, sizeof(int));
    val++;
    lseek(fd, 0, SEEK_SET);
    write(fd, &val, sizeof(int));
    fl.l_type = F_UNLCK;
    fcntl(fd, F_SETLK, &fl);
```

Cliente - Servidor

Cliente

```
int main (int argc, char *argv[]){
  int cliente_id;
  struct sockaddr_in dir_cliente;
  unsigned char byte;
  bzero((char*) &dir_cliente, sizeof(dir_cliente));
  dir_cliente.sin_family = AF_INET;
  dir_cliente.sin_port = htons(7);
  cliente_id = socket(AF_INET, SOCK_STREAM, IPPORT_TCP);
  connect(cliente_id, (struct sockaddr*) &dir_cliente, sizeof(dir_clie)
  while (read(0, &byte, 1) == 1){
    send(cliente_id, &byte, 1, 0);
    recv(cliente_id, &byte, 1, 0);
    write(1, &byte, 1);
  }
  close(cliente_id);
}
```

Servidor

```
int main (int argc, char* argv[]){
  int server_id, cuenta_id, tam;
  unsigned char byte;
  struct sockaddr_in server_dir, cliente_dir;
  server_id = socket(AF_INET, SOCK_STREAM, IPPORT_TCP
  bzero((char*) &server_dir, sizeof(server_dir));
  server_dir.sin_family = AF_INET;
  server_dir.sin_port = htons(7);
  server_dir.sin_addr.s_addr = INADDR_ANY;
  bind(server_id, (struct sockaddr*) &server_dir,
       sizeof(server_id));
  listen(server_id, 100)
  while(1){
    tam = sizeof(cliente_dir);
    cliente_id = accept(server_id, (struct sockaddr*)
         &cliente_dir, sizeof(cliente_dir));
    switch(fork()){
      case 0:
        close(server_id);
        while(recv(cliente_id, &byte, 1, 0) ==1)
            send(cliente_id, &byte, 1, 0);
        close(cliente_id);
        return 0;
      default:
        close(cliente_id);
    }
  }
```

Sistemas Operativos. Examen extraordinario. 6-julio-2015

(*Publicación de notas: **14-julio**. Revisión: **16-julio, 12h**.*)

Dispone de 2 HORAS.

(*Al terminar debe entregar las hojas en 2 montones: uno para Teoría 1 y Ejercicio 1, y otro para Teoría 2 y Ejercicio 2*)

TEORÍA 1 de 2 (2 puntos)

T1) [1 pto] Supuesto que tengamos un disco de 1TB con un sistemas de ficheros tipo UNIX, con agrupaciones de tamaño 8KiB e i-nodos de 128B. Se pretende averiguar en qué se emplean los datos y los metadatos, para ello responda a las siguientes preguntas:

- Indique las partes en que se organiza el sistema de ficheros, explicando qué es datos y qué es metadatos.
- Queremos llenar el disco de ficheros de un tamaño medio de 10MB, ¿Cuántos i-nodos necesitaríamos?
- Calcule el espacio que ocupan los mapas de bits:
  - Mapa de bits de agrupaciones para todo el disco.
  - Mapa de bits de i-nodos.
- ¿Despreciando el espacio reservado para el sector de arranque y el superbloque, ¿cuántas agrupaciones de datos quedarían disponibles en el disco?

T2) [0,5 pto] Explique en qué consiste el mecanismo de optimización copy-on-write (COW). Ponga un ejemplo describiendo dicha optimización para el caso que invoquemos al servicio fork().

T3) [0,5 pto] Explique qué es la hiperpaginación o trashing,

- ¿Qué es y por qué se produce?
- ¿Cómo afecta al funcionamiento de un computador?
- ¿Cómo se resuelve?

EJERCICIO 1 (3 puntos)

a) [1 pto] Escriba el programa "mi_copia fichero1 fichero2" que copia el fichero1 en el fichero2. Haga el programa lo más sencillo posible, sin necesidad de hacer tratamiento de errores y haciendo las suposiciones que estime oportunas.

Ejecutamos el programa anterior. Se pretende rellenar en la siguiente tabla las distintas características de las regiones del mapa de memoria del proceso. Se pone a modo de ejemplo la región de código.

Nombre de la región	Permisos de la región	Procedencia del contenido	¿Es necesario rellenar de 0?	Región compartida o privada	Ubicaciones posibles de las páginas	¿La región puede crecer ?	Descripción de la región
Código	R,X	Del fichero ejecutable	No, se rellena con el programa del ejecutable	Compartida	Páginas en memoria o en el fichero ejecutable	No	Incluye el código del fichero ejecutable
...							

b) [1 pto] Complete dicha tabla añadiendo en cada fila una de las siguientes regiones y sus características:

- (1) Variables globales, con valor inicial, (2) variables globales sin valor inicial, (3) heap, (4) biblioteca dinámica, (5) variables globales de la biblioteca dinámica y (6) pila

c) [0,5 pto] En el programa principal se invoca a; "`int función(int a, char c)`". Describa en detalle, ayudándose de un dibujo, el contenido de la pila antes y durante la invocación a dicha función. Suponga la función principal "`int main(int argc, char **argv)`".

d) [0,5 pto] Supuesto que se desea proyectar en memoria el fichero "`/tmp/fichero.txt`", programe una línea para proyectarlo, otra para desproyectarlo y un breve ejemplo de acceso en memoria al contenido del fichero, tanto en lectura como en escritura.

TEORÍA 2 de 2 (2 puntos)

T4) [1 pto] Explicar cómo un programa puede poner en ejecución un fichero ejecutable y obtener su código de terminación. (Citar los nombres de los servicios necesarios)

T5) [1 pto] Explicar cómo conseguir que un proceso pueda estar preparado desde el primer instante de su ejecución para realizar una acción si recibe una determinada señal. Por ejemplo, si la señal llega antes de que la primera sentencia del código fuente se haya ejecutado.

EJERCICIO 2 (3 puntos)
Dado el siguiente programa:

a) [0,5 pto] ¿Qué escribe en la salida estándar?

b) [0,5 pto] Si sustituimos "PTHREAD_CREATE_JOINABLE" por "PTHREAD_CREATE_DETACHED", y eliminamos el bucle "A" al completo, ¿Puede cambiar el valor final de n? ¿Porqué?

c) [0,5 pto] Si sustituimos "pthread_cond_broadcast" por "pthread_cond_signal" (ver línea B), ¿Puede cambiar el valor final de n? ¿Porqué?

d) [0,5 pto] Si eliminamos las líneas C1 y C2, ¿Puede cambiar el valor final de n? ¿Porqué?

e) [0,5 pto] Si en la línea D sustituimos "while" por "if", ¿Puede cambiar el valor final de n? ¿Porqué?

f) [0,5 pto] Si eliminamos las líneas C1, C2, D, E1 y E2, ¿Puede cambiar el valor final de n? ¿Porqué?

```
#define N 7
int n = 0;
int e1 = 0;
pthread_mutex_t m1;
pthread_cond_t c1;

void *f1(void *p){
      int i;
      pthread_mutex_lock(&m1); /*E1*/
      while(e1) pthread_cond_wait(&c1, &m1); /*D*/
      e1 = 1;
      pthread_mutex_unlock(&m1); /*C1*/
      n = n + 1;
      pthread_mutex_lock(&m1); /*C2*/
      e1 = 0;
      pthread_cond_broadcast(&c1); /*B*/
      pthread_mutex_unlock(&m1); /*E2*/
      pthread_exit(NULL);
}

int main(void){
      int i;
      pthread_t v[N];
      pthread_attr_t attr;
      pthread_attr_init(&attr);
      pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);
      for(i = 0; i < N; i++)
            pthread_create(&v[i], &attr, &f1, NULL);
      for(i = 0; i < N; i++) /*A*/
            pthread_join(v[i], NULL);
      pthread_attr_destroy(&attr);
      printf("n = %i\n", n);
      return 0;
}
```

Sistemas Operativos 4º Semestre. Grados II y MI

Primer Parcial. Sistema de Ficheros. 12 de Marzo de 2015.

Dispone de 60 minutos. Publicación: el 26 de Marzo. Revisión: el 10 de Abril.

Ejercicio único

Los siguientes apartados están relacionados.

A) (2 puntos) Implemente en C y para UNIX el mandato `incrementa` *`archivo`*, que:

A1) Abre el archivo indicado como argumento y redirige la entrada y la salida estándar a dicho archivo.

A2) Seguidamente, a través de los descriptores estándar, incrementa en 5 el entero binario (`int`) contenido en la posición 3 GiB del archivo.

SOLUCIÓN:

A1)

```
1   int main(int argc,char*argv[])
2   {
3       int fch = open(argv[1],O_RDWR|O_CREAT,0666);
4
5       dup2(fch,0); // fch es el 0
6       dup2(fch,1); // fch es el 1
7       close(fch);
```

A2)

```
8   #define POS (3UL*(1<<30))
9       int ofs = lseek(0,POS,SEEK_SET);
10
11      int val = 0;
12      int cnt = read(0,&val,sizeof(int)); // Lee el valor
13
14      ofs = lseek(1,-cnt,SEEK_CUR); // Coloca antes del valor
15
16      val += 5;
17      cnt = write(1,&val,sizeof(int)); // Escribe el nuevo valor
18
19      return 0;
20  }
```

B) (2 puntos) Según su implementación, ¿cómo se comportaría `incrementa` en los siguientes casos?:

B1) Si el archivo indicado fuese un fichero especial orientado a carácter, (ej. un terminal).

B2) Si el archivo indicado fuese un fichero normal con tamaño inicial 2 MiB. ¿Qué tamaño al final?

3GiB + int

SOLUCIÓN:

B1) Suponiendo que el terminal pudiese ser abierto, las llamadas `lseek` fallarían (no existe el concepto de "posición" sobre un terminal) y las llamadas `read` y `write` estarían dialogando a través del terminal con un usuario, que no puede introducir información en binario, sino sólo texto.

B2) La primera llamada `lseek` se posicionaría más allá del tamaño del fichero, con lo que la llamada `read` devolvería `0` bytes leídos. Tal y como se ha implementado, `val` valdría `0`, la segunda llamada a `lseek` no nos movería de la posición `POS` y la llamada `write` escribiría el entero binario de valor `5` en la posición indicada. El tamaño final del fichero serían `POS` más `sizeof(int)` bytes, y el espacio intermedio desde los 2 MiB hasta los 3 GiB se habría rellenado con bytes nulos.

C) (2 puntos) Considerando que la cache de bloques del servidor de ficheros está inicialmente vacía y utilizando los mismos datos y bajo las mismas suposiciones del apartado **D**, conteste a las siguientes preguntas, donde nos referimos a las llamadas necesarias para implementar el mandato del apartado **A** (que incrementa el entero binario contenido en la posición 3 GiB de un archivo).

C1) Razone, ¿cuántos accesos a disco son necesarios para realizar la llamada `read` (del apartado **A**)?

C2) Razone, ¿cuántos accesos a disco son necesarios para realizar la llamada `write` (del apartado **A**)?

SOLUCIÓN:

C1) Para acceder a la posición 3 GiB de este archivo habrá que acceder a la agrupación lógica número (3*2^30 (B) / 2^13 (B/agrp) = 3*2^17 (agrp)) 393216 de su inodo.
Según las suposiciones realizadas en el apartado D (direcciones de 32 bits y 2^11 (dirs/argp)), habrá que acceder a través del puntero doble indirecto del inodo. Esto supone acceder a tres agrupaciones. Como la cache de bloques está inicialmente vacía, serán accesos al disco.

C2) Para la correspondiente operación de escritura sobre la misma posición del archivo, dado que la agrupación afectada (así como las agrupaciones de indirección necesarias para localizar esta) ya estarán en la cache de bloques, no será necesario acceder al disco, bastará con realizar esta modificación sobre la cache de bloques, esto es, en memoria. La posterior escritura desde la cache hacia el disco de esta información contenida en la cache y ahora sucia (agrupación con número lógico 393216) sucederá más tarde, dependiendo de la política de gestión de esta cache.

D) (2 puntos) Considere que el sistema de ficheros subyacente es de tipo UNIX (basado en inodos), con agrupaciones de 8 KiB, para un disco de medio TiB y archivos de tamaño medio igual a una agrupación. Justifique las suposiciones que considere necesarias.

D1) Exprese la capacidad de direccionamiento de un inodo en este sistema. Dé el resultado en bytes.

D2) Razone, ¿cuánto ocupará el mapa de bits de inodos en este sistema? Indique cálculos y unidades.

SOLUCIÓN:

D1) Lo primero hace falta saber es el tamaño de las direcciones a agrupación.
Dado que: 2^39 (B/disco) / 2^13 (B/agrp) = 2^26 (argp/disco), bastan 26 bits para numerar todas las agrupaciones, luego usaremos direcciones de 32 bits.
En una agrupación de indirección cabrán: 2^13 (B/agrp) / 2^2 (B/dir) = 2^11 (dirs/argp)
Suponiendo un inodo con 10 punteros directos y 3 niveles de indirección, se podrán direccionar:
[10 (p.dir) + 2^11 (s.ind) + 2^22 (d.ind) + 2^33 (t.ind)] (agrupaciones).
Para pasar a bytes multiplicamos por el tamaño de la agrupación en bytes.
En números redondos serían 2^(33+13) (B) = 64 TiB y pico.

D2) Preparamos un inodo por cada posible archivo de tamaño medio, esto es, 2^26 (inodos).
El mapa de bits necesita un bit por inodo, luego ocupará: 2^26 (bits) / 2^3 (bits/B) / 2^13 (B/agrp) = 2^10 (agrupaciones).

E) (2 puntos) Sea la siguiente visión parcial del contenido del sistema de ficheros:

NUM	T	U	G	O	USER	GROUP	PATH (RUTA)
1	d	rwx	r-x	r-x	root	root	/
2	d	rwx	rwx	rwt	root	root	/tmp/
3	-	rwx	rw-	r--	yoda	jedi	/tmp/datos.bin
4	d	rwx	r-x	r-x	root	root	/home/
5	d	rwx	r-x	---	r2d2	robot	/home/r2d2/
6	l	rwx	rwx	rwx	r2d2	robot	/home/r2d2/temporal -> /tmp/
7	-	rwx	r-s	---	r2d2	jedi	/home/r2d2/incrementa

Donde: NUM es el número de ruta; T es el tipo de objeto; U, G y O son los grupos de permisos (*user*, *group* y *other*); y la notación A -> B indica que el enlace simbólico A apunta a la ruta B.

Considere que el usuario r2d2 (del grupo robot) invoca, desde su *home*, el siguiente mandato:

```
./incrementa  temporal/datos.bin
```

Para cada uno de los siguientes casos, conteste detallando cómo el sistema operativo decodificará la ruta indicada. Muestre: (número de) ruta, grupo de permisos y permiso concreto, que se validará a cada paso.

E1) ¿Podrá ponerse en ejecución el mandato indicado? ¿Qué identidad tendrá el proceso resultante?

E2) ¿Podrá el proceso abrir el archivo indicado? Detalle cada paso de la decodificación de la ruta.

SOLUCIÓN:

E1) Sí podrá ejecutarse la ruta relativa: ./incrementa
Detalle: [./ 5Ux] [incrementa 7Ux].
Tendrá identidad efectiva de grupo jedi, por [incrementa 7Gs].

E2) Sí podrá abrirse la ruta relativa: temporal/datos.bin
Detalle: [./ 5Ux] [temporal/ 6Tl] (seguimos con /tmp/) [/ 1Ox] [tmp/ 2Ox (la x bajo la t)] (seguimos con datos.bin) [datos.bin 3Grw].

Apellidos, Nombre: _____ Nº Matrícula: _____ D.N.I.: _____

Sistemas Operativos – 4º Semestre – GII & GMI

Segundo Parcial. Gestión de Procesos. 20 de abril de 2015

Dispone de 60 minutos. Las notas saldrán el Martes 12 de mayo. La revisión será el Jueves 14 de mayo a las 9:00 en la sala Multiusos del DATSI (S4200, Vestíbulo de la planta 2ª del Bloque IV)

Cuestiones (conteste la cuestión 1 en el espacio reservado al efecto en esta misma hoja y el resto en una hoja aparte)

1) [1 pto] Suponga que en un sistema monoprocesador y mononúcleo multitarea se han creado 3 procesos de usuario denominados PA, PB y PC. Suponiendo que los 3 procesos se encuentran en la cola de procesos listos para ejecutar, represente en el cronograma de abajo los estados por los que pasan TODOS los procesos involucrados con la siguiente secuencia:

a) El proceso PB ejecuta durante 2 unidades de tiempo e invoca un servicio del sistema operativo.

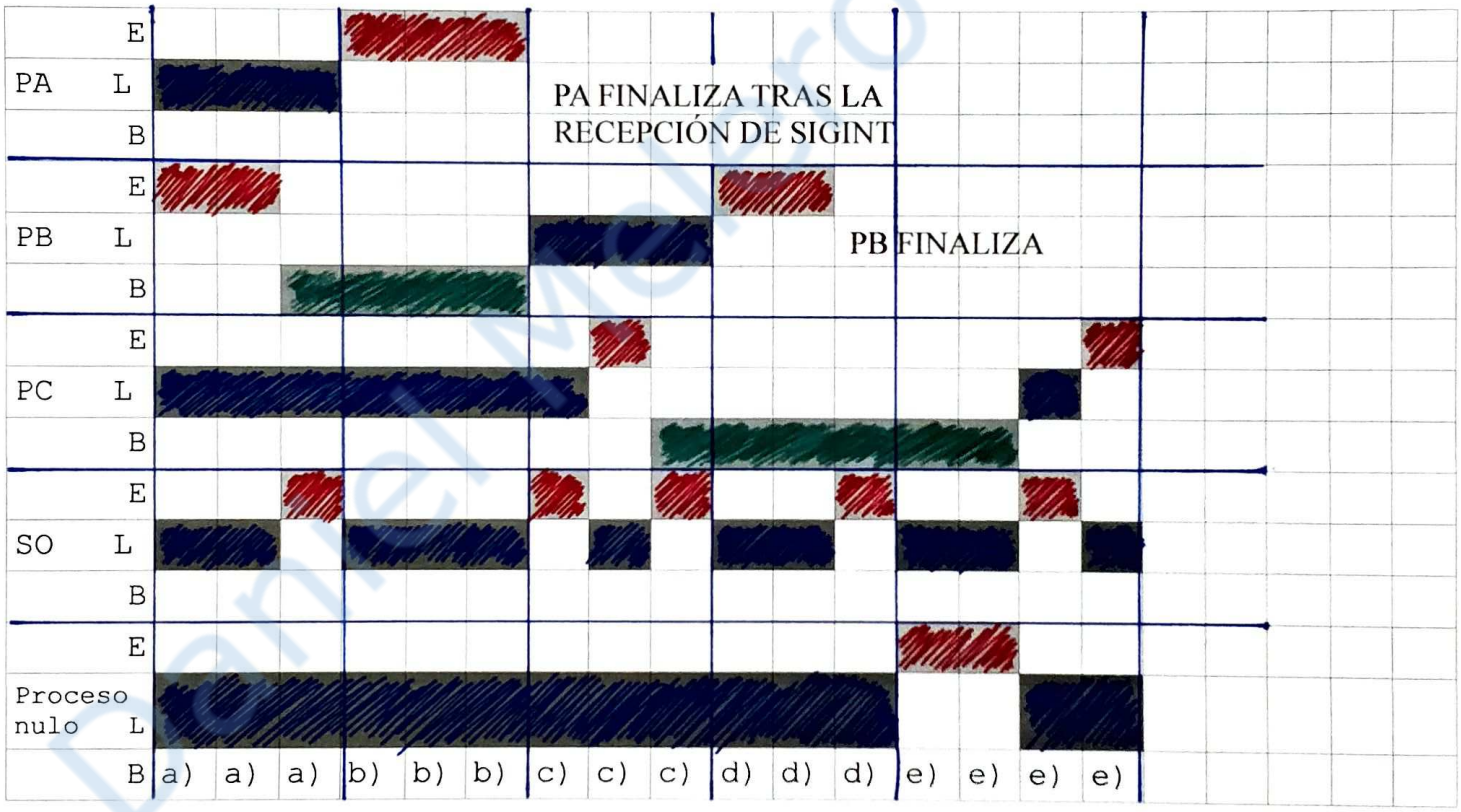
b) El proceso PA entra en ejecución y al cabo de 3 unidades de tiempo recibe del sistema operativo la señal SIGINT con el tratamiento por defecto.

c) Se completa el servicio demandado por el proceso PB y entra en ejecución el proceso PC durante 1 unidad de tiempo, momento en el que PC solicita una operación de E/S.

d) El proceso PB termina tras 2 unidades de tiempo.

e) La situación no cambia hasta transcurridas 2 unidades de tiempo, tras las cuales el disco duro genera una interrupción indicando que se ha completado la operación de E/S pendiente.

En los casos en los que no se haya especificado, suponga que los procesos involucrados intervienen durante 1 unidad de tiempo.



Durante su ciclo de vida, los procesos pueden estar en 3 estados diferentes: ejecución, listo para ejecutar o bloqueado.
Notación: Rojo-Proceso ejecutando; Azul-Proceso listo para ejecutar; Verde-Proceso bloqueado
En un momento determinado, siempre tiene que estar en ejecución un único proceso.
El tratamiento por defecto con la recepción de la señal SIGINT señal provoca la muerte del proceso que la recibe.
Cuando no hay procesos de usuario listos para ejecutar, entra en ejecución el proceso nulo.

2) [1 pto] Indique la arquitectura software de una aplicación diseñada con *threads* que resulta equivalente a la arquitectura de un servidor paralelo implementado con procesos pesados *monothread*.

La formada por un thread distribuidor y varios threads trabajadores creados por cada solicitud de servicio.

3) [1 pto Indique las causas por las que el servicio `kill` puede finalizar con error.

Se ha especificado una señal inválida, el proceso no tiene permiso para enviar la señal a ninguno de los procesos destinatarios o el proceso o los procesos destinatarios no existen.

4) [2 ptos] Especificar el código necesario para que un proceso compruebe si el hijo con identificador de proceso `pid` ha finalizado por la recepción de una señal o por otra causa, imprimiendo en cualquier caso el valor de finalización del hijo si este valor es distinto de 0.

```
#include <errno.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
pid_t pid, aux; int status;

pid=fork();
...
/* En el proceso padre */
do {
   aux=wait(&status);
while ((aux!=pid) && (aux > 0));
/* Otra posibilidad en lugar del bucle: aux=waitpid(pid,&status,0); */
if (aux < 0) { /* Comprobación de error */
   perror("wait"); exit(1); }
if (WIFEXITED(status))
   printf ("Hijo finaliza con código: %d\n", WEXITSTATUS(status));
if (WIFSIGNALED(status))
   printf("Hijo finaliza con señal: %d\n", WTERMSIG(status));
```

Problema 1 (Solución)

a) **[1 pto]** Complete el código de un servidor que atiende las solicitudes de los clientes a través del descriptor de fichero `fpc`. Suponga que el servidor recibe la información a procesar disponiendo de suficiente espacio de almacenamiento para atender las solicitudes de los clientes con una sola invocación del servicio `read`. Una vez que se tienen los datos disponibles, el servidor los escribirá en el descriptor de fichero `fd` (`fd` ya abierto y asociado al dispositivo gestionado por el servidor). Se ha especificado con comentarios la parte correspondiente a la comunicación inicial con los clientes (apertura del descriptor de fichero `fpc` y recepción de la orden del cliente).El resto de operaciones involucradas en el servicio, recepción y escritura de los datos de la solicitud y contestación al cliente a través de `fpc`, se han especificado directamente en el código.
En este apartado sólo deben completarse los posibles errores en la invocación de los servicios del sistema.

```
/* Las únicas llamadas a servicios del sistema son read y write.  */
#define TAM_MAX 8*1024*1024 //Tamaño del espacio de almacenamiento
#define SERVICIO_OK "PETICION COMPLETADA CON EXITO"
#define SERVICIO_NO_OK "PETICION COMPLETADA SIN EXITO"
int main(int argc, char **argv){
int fd, fpc, n_leidos, n_escritos, n_contestados;
char buffer[TAM_MAX];
/* Código que abre y configura el puerto de comunicación para recibir las solicitudes de los
clientes */
while (1) {
      /* Recepción de la orden del cliente */
      n_leidos = read(fpc, buffer, TAM_MAX);
      if (n_leidos == -1)
            perror("Error en la lectura de los datos");
      /* Ejecución de la petición solicitada */
      n_escritos = write(fd, buffer, n_leidos);
      if (n_escritos == -1)
            perror("Error en la escritura de los datos");
      else if (n_leidos != n_escritos){
            fprintf(stderr, "Se han escrito menos datos de los leidos\n");
            strcpy(buffer, SERVICIO_NO_OK);
      }
      else strcpy(buffer, SERVICIO_OK);
      /* Contesta al cliente el resultado del servicio realizado */
      n_contestados = write(fpc, buffer, strlen(buffer));
      if (n_contestados == -1)
            perror("Contestación incorrecta);
      else if (n_contestados != strlen(buffer))
            fprintf(stderr,"Contestación incompleta\n");
      }
}
```

En el código de este servidor no procede invocar el servicio `exit` o la sentencia `return` con este tipo de errores porque el proceso servidor terminaría, dejando de dar servicio a los clientes.

b) **[1 pto]** Optimice el código del apartado a) implementando un servidor paralelo con procesos pesados (procesos hijo). Los procesos hijo que se vayan creando deberán responder las solicitudes de los clientes, mientras que el proceso padre, por su parte, deberá evitar la aparición de procesos zombie.
Nota importante: para la implementación de la parte del proceso padre se deberá usar la llamada `waitpid(-1,&status,WNOHANG);`
que devuelve el pid del proceso que haya finalizado, pero que no se queda bloqueada (opción WNOHANG) en caso de que no haya finalizado ninguno de los procesos hijo y en este caso devuelve un 0.

```
/* Mismo código del apartado a) del que solo se muestran los comentarios */
/* Se añaden las siguientes variables locales de la función main */
pid_t pid; int status;
while (1) {
   /* Recepción de la orden del cliente */
   /* Si la invocación del read se deja en el hijo, no habría ningún tipo de
   control respecto del número de procesos hijo creados, pudiendo provocar que la
   tabla de procesos se colapse. De esta forma solo se crea uno por cada solicitud
   de los clientes. El resto del código sí puede ejecutarlo el hijo. */
      pid=fork(); /* Creación del hijo solo tras la recepción de la orden. */
      if (pid == -1) {
         perror("Error en la creación del hijo"); exit(2); }
      else if (pid ==0) {
               /* Ejecución de la petición solicitada */
               /* Contesta al cliente el resultado del servicio realizado */
               n_contestados=write(fpc,buffer,strlen(buffer));
               if (n_contestados == -1){
                     perror("Contestación incorrecta); exit(3); } /* Cuando se
   produce un error en la ejecución del hijo, no tiene sentido seguir con su
   ejecución y debe finalizar. */
               else if (n_contestados != strlen(buffer)) {
                     fprintf(stderr,"Contestación incompleta\n"); exit(4);}
               exit(0);
      }
      else pid=waitpid(-1,&status,WNOHANG); /* Con WNOHANG el padre no se queda
      bloqueado esperando la finalización de cualquiera de sus hijos si éstos no
      han terminado. Se devuelve -1 cuando no hay más hijos. Para recoger el
      estado de los hijos, el proceso servidor debería armar la señal SIGCHLD,
      pero no se ha considerado esta parte en esta solución. */
   }
```

Otra solución posible:

```
/* Mismo código del apartado a) del que solo se muestran los comentarios */
void tratar_fin_hijos(void){
   int status;
   waitpid(-1,&status,WNOHANG);
   /* Verificación del estado de finalización del hijo */
}
int main(int argc, char **argv){
struct sigaction act; pid_t pid;
…
act.sa_handler=&tratar_fin_hijos;
act.sa_flags=SA_RESTART;
sigaction(SIGCHLD,&act,NULL);
while (1) {
   /* Recepción de la orden del cliente */
      pid=fork(); /* Creación del hijo solo tras la recepción de la orden. */
      if (pid == -1) {
         perror("Error en la creación del hijo"); exit(2); }
      else if (pid ==0) {
               /* Ejecución de la petición solicitada */
               /* Contesta al cliente el resultado del servicio realizado */
               n_contestados=write(fpc,buffer,strlen(buffer));
               if (n_contestados == -1){
                     perror("Contestación incorrecta); exit(3); }
               else if (n_contestados != strlen(buffer)) {
                     fprintf(stderr,"Contestación incompleta\n"); exit(4);}
               exit(0);
      }
   }
```

c) **[1 pto]** Controle la duración de la ejecución de cada hijo de modo que transcurridos 10 segundos sin haber conseguido enviar la información por `fd` muestre por salida estándar de error el mensaje de aviso "No se ha lanzado el servicio tras 10 segundos". Transcurridos otros 30 segundos más sin que se haya enviado la información, se mostrará por la salida estándar de error "Trabajo rechazado" y el proceso hijo deberá finalizar.

```
int alarma_10_segundos; /* Variable global */
void tratar_alarma(void) {
      if (alarma_10_segundos) {
            /* Activación del temporizador de 30 segundos */
            alarma(30);
            alarma_10_segundos=0;
            fprintf(stderr,"No se ha lanzado el servicio tras 10 segundos\n");
            return 0;}
      else { /* Vencimiento de la alarma de 30 segundos y el hijo finaliza
devolviendo al padre el estado en el que ha terminado. */
            fprintf(stderr,"Trabajo rechazado\n"); exit(5);}
}
struct sigaction act; /* Variable local de la función main */
/* Mismo código que en el apartado anterior */
while (1) {
       /* Lectura de la orden del cliente */
          pid=fork();
          if (pid == -1) {
             perror("Error en la creación del hijo"); exit(3);
          else if (pid ==0) {
             alarma_10_segundos=1; /* Distingue entre temporizar 10 ó 30 segundos */
             /* Armado de la señal SIGALRM solo en los hijos. */
             act.sa_handler=&tratar_alarma;
             act.sa_flags = SA_RESTART; /* evita E_INTR en el wait, read y write */
             sigaction(SIGALRM, &act, NULL);
             alarm(10); /* Activación del temporizador de 10 segundos */
             /* Mismo código del bucle especificado en el apartado anterior */
             ...
             n_escritos=write(fd,buffer,n_leidos);
             alarm(0);/* Se ha conseguido lanzar el servicio y por lo tanto se
             desactiva el temporizador. Si no se desactiva, el proceso hijo podría
             finalizar por el vencimiento de alguno de los temporizadores mientras
             que está procesando la solicitud del cliente. */
             ...
          }
```

d) **[2 ptos]** Repita el apartado b) suponiendo que los procesos independientes se ejecutan como threads. El servidor sólo finalizará su ejecución si se produce un error en la ejecución del hilo principal.

```
int aux;
/* Las variables de los descriptores de ficheros se definen como globales para que
sean visibles en los threads */
int fd, fpc;

/* Función que se ejecuta en cada uno de los hilos que dan servicio a las
   solicitudes de los clientes. Ejecuta el mismo código que antes ejecutaban los
   hijos salvo la invocación de exit, que se sustituye por return o pthread_exit
*/

void *atiende(){
int n_escritos, n_contestados;
char buffer[TAM_MAX];

/* Ejecución de la petición solicitada */
n_escritos=write(fd,buffer,n_leidos);

if (n_escritos == -1) {
      perror("Error en la escritura de los datos"); return 2;} * Cuando se produce un
      error en la ejecución del thread, no tiene sentido seguir con su ejecución y
      debe finalizar, con return o con pthread_exit. */
else if (n_leidos != n_escritos){
         fprintf(stderr,"Se han escrito menos datos de los leidos\n");
         strcpy(&buffer,SERVICIO_NO_OK);
         }
   else strcpy(&buffer,SERVICIO_OK);

/* Contesta al cliente el resultado del servicio realizado */
n_contestados=write(fpc,buffer,strlen(buffer));
if (n_contestados == -1) {
     perror("Contestación incorrecta); return 5; }
else if (n_contestados != strlen(buffer)) {
      fprintf(stderr,"Contestación incompleta\n"); return 6; }
pthread_exit(0);
}

int main(int argc, char **argv){
/* Mismo código que en el apartado a) */
pthread_t thid;  /* Solo se utiliza para recoger el identificador de thread en la
creación de los threads. */
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED); /* Independientes para
que tenga la misma funcionalidad que un servidor paralelo con procesos pesados
(apdo. b). */

while (1) {
      /* Recepción de la orden del cliente */
      n_leidos=read(fpc,&buffer,TAM_MAX);
      if (n_leidos == -1){
         perror("Error en la lectura de los datos"); return 1; } /* Error en el hilo
         principal. El servidor debe finalizar según lo especificado en el
         enunciado. */
      /* Creación del thread solo tras la recepción de la orden del cliente. */
      aux=pthread_create(&thid,&attr,&atiende,NULL);
      if (aux == -1) {
         perror("Error en la creación del thread"); exit(2); } /* Error en el hilo
         principal. El servidor debe finalizar según lo especificado en el
         enunciado. */
      }
}
```

Sistemas Operativos – 4º Semestre – GII & GMI

Tercer Parcial. Gestión de Memoria. 18 de mayo de 2015

Sea un sistema Linux con páginas de 2 KiB, palabras y direcciones de 32 bits, 3 GiB de RAM y un disco de 1 TiB. El fichero /home/pepe/datos.bin tiene un tamaño de 1MiB y está relleno con los enteros en binario 0, 1, 2, 3, 4, 5, etc.

```
#include <stdio.h>
#include <pthread.h>
int fdin, fdout, a, *p, *q; ...     //hay más declaraciones
void* fun1 (void* arg)
{...}
void* fun2 (void* arg)
{...}
int main(void) {
   pthread_t fun1th, fun2th;
   struct stat mistat;
   fdin = open("/home/pepe/datos.bin",O_RDWR);
   fstat(fdin, &mistat);
   p = mmap(0,mistat.st_size,PROT_READ|PROT_WRITE,MAP_PRIVATE,fdin,0);
   close(fdin);
   pthread_create(&fun1th, NULL, fun1, NULL);
   pthread_create(&fun2th, NULL, fun2, NULL);
   pthread_join(fun1th, NULL);
   pthread_join(fun2th, NULL);
   q = (int *) *p;
   *q = 4;
   a = *q;
   munmap(p, mistat.st_size);
   return 0;
}
```

Se pide:

***a) (2 ptos)** Razonar si los dos threads llegan a compartir la región creada por el mmap.*

Todo proceso pesado se crea (mediante un `fork`) con un único flujo de ejecución inicial, que denominamos hilo principal y cuya pila es la pila del proceso. Cualquier otro hilo (proceso ligero o *thread*) creado (mediante un `pthread_create`) desde el proceso, pasa a formar parte del proceso que lo crea. Cada nuevo hilo precisa un espacio de memoria (normalmente tomado del heap) para ser usado como su pila, pero todos los hilos del proceso (incluido el principal) son una unidad desde el punto de vista del Gestor de Memoria del Sistema Operativo. La imagen de memoria de cada proceso pesado concreto es una única para todo el proceso, independientemente de cuantos hilos tenga dicho proceso. De hecho, la tabla de páginas que da soporte a la imagen de memoria de tal proceso es única y se mantiene en la MMU mientras se planifique cualquiera de los hilos del proceso. Es por esto que el cambio de contexto entre los hilos de un proceso es mucho más rápido que entre hilos de procesos distintos.

Dicho esto, es evidente que todos los hilos del proceso comparten la misma imagen de memoria y que cualquier alteración que un hilo haga en dicha imagen la ven los demás. Por lo tanto la zona proyectada en este ejercicio es compartida por los tres hilos del proceso. El flag `MAP_PRIVATE` tan sólo implica que los cambios que se realicen en esta región proyectada no sean permanentes en el fichero ni sean visibles desde otros procesos.

b) (2 ptos) *Indicar razonadamente el máximo tamaño que podría tener la imagen de memoria de un proceso.*

La imagen de memoria de un proceso es el conjunto de regiones de memoria asignadas a un proceso. Esta imagen está soportada sobre el mapa de memoria del sistema. Si el sistema sólo dispone de memoria real el mapa de memoria estará soportado sólo sobre la memoria principal, pero si el sistema dispone de memoria virtual podrá estar soportado además sobre parte del almacenamiento secundario: área de intercambio (*swap*) y ficheros proyectados.

El tamaño máximo del mapa de memoria es el rango de direcciones que el procesador puede producir. En un sistema de 32 bits, como el del ejercicio, pueden direccionarse 2^32 bytes (no bits, ni palabras ni mucho menos páginas). Usualmente una parte fija de este mapa se reserva para el contener el propio núcleo (*kernel*) del Sistema Operativo más el espacio para datos que éste pueda necesitar. Esta parte del mapa estará marcada para residir permanentemente en memoria principal y así se indica en las tablas de páginas de todos los procesos. De esta manera, el cambio de modo usuario a modo kernel y viceversa (cada vez que se llame al sistema) será rápido. Por supuesto, este rango del mapa de memoria para la parte residente del sistema operativo no estará disponible para los procesos y por lo tanto limitará el tamaño máximo de la imagen de memoria de los procesos. En este ejercicio supondremos que el núcleo precisa 1 GiB residente.

Si el sistema no dispusiera de memoria virtual, la imagen de memoria de todos los procesos debería convivir en el espacio de memoria principal restante, esto es, 2 GiB. Este sería el límite máximo.

Pero si disponemos de memoria virtual con espacio virtual separado por proceso, el tamaño máximo de la imagen de memoria de un proceso serían 3 GiB, que estarían soportados sobre los 2GiB de marcos de página disponibles más el espacio de disco destinado a área de intercambio y ficheros proyectados.

c) (2 ptos) *Suponiendo que los threads no modifican ni p ni la región del mmap, indicar el valor que tendrá la variable* ***a*** *antes del munmap.*

Con el primer argumento de la llamada `mmap` a 0 se le pide al sistema que escoja el rango de direcciones que considere más conveniente para proyectar el archivo. Así `p` valdrá una dirección a partir de la cuál se encontrará proyectado el contenido del archivo, con los valores 0, 1, 2, etc. como enteros de 32 bits.

La sentencia "`q = (int *) *p;`" toma el valor apuntado por `p` (el 0), lo interpreta como puntero a entero, y se lo asigna a `q`. Por lo tanto `q` es un puntero a entero de valor 0. En otras palabras, `q` vale `NULL`.

La sentencia "`*q = 4;`" pretende asignar un valor 4 en la dirección apuntada por `q`, pero como `q` vale `NULL`, eso implica dereferenciar un puntero nulo y eso no se puede hacer y para ello la dirección 0 está protegida, para que salte un `SIGSEGV`.

En este momento el proceso moriría y la siguiente sentencia "`a = *q;`"no llegaría a ejecutarse.

d) (4 ptos) *Teniendo en cuenta los siguientes tamaños y que se utiliza montaje dinámico al invocar el procedimiento, rellenar la tabla adjunta con la información de la imagen de memoria del proceso antes de que terminen los dos threads.* ***Utilizar una línea por cada región****, rellenando* ***todos los campos*** *de la tabla.*

`text`	`data`	`bss`	`filename`
`57653 = 56 KiB + 309 B`	`273 B`	`374`	`Miprogr.o`
`1719061 = 1678 KiB +789 B`	`11508 = 11 KiB + 244 B`	`11316 = 11 KiB + 52 B`	`libc.so`
`92630 = 90 KiB + 470 B`	`868 B`	`8352 = 8KiB + 160 B`	`libpthread.so`

Consideraciones generales:

- La Dirección inicial y el Tamaño deben ser valores múltiplos del tamaño de la página (2 KiB).
- La Dirección final indica la última dirección con contenido de la región según lo especificado en el enunciado.
- La presentación de las regiones ha sido según se han ido creado durante la ejecución del proceso. Las pilas aparecen en las posiciones más altas de la imagen de memoria del proceso.
- En este ejercicio no ha sido necesario crear una región específica para heap, pero también se considera válido crearla sin que lo demande el proceso o agruparla junto con las regiones de datos del proceso.

Región	Dirección inicial	Dirección final	Tamaño	Características	Origen del contenido de la región
SSOO	0	9.437.184	9 MiB, 4.608 páginas	---	---
...	...	...	...	...	...
Código Miprogr.o	10.485.760 (10 MiB)	10.543.412	58 KiB, 29 páginas	R-X, compartida, tamaño fijo	Ejecutable en el sistema de ficheros
DVI Miprogr.o	10.545.152 (10 MiB + 58 KiB)	10.545.424	2 KiB, 1 página	RW-, privada, tamaño fijo	Ejecutable en el sistema de ficheros
DSVI Miprogr.o	10.547.200 (10 MiB + 60 KiB)	10.547.573	2 KiB, 1 página	RW-, privada, tamaño variable (heap se incluye aquí; si no, sería de tamaño fijo)	Rellenar con ceros el marco de página asignado
...	...	...	...	...	...
Pila Migprogr.o	18.874.368 (18 MiB)	18.771.969	100 KiB, 50 páginas	RW-, privada, tamaño variable	Rellenar con ceros el marco de página asignado
...	...	...	...	...	...
Código libc.so	10.549.248 (10 MiB + 62 KiB)	12.268.308	1680 KiB, 840 páginas	R-X, compartida, tamaño fijo	Biblioteca en el sistema de ficheros
DVI libc.so	12.269.568 (11 MiB + 718 KiB)	12.268.308	12 KiB, 6 páginas	RW-, privada, tamaño fijo	Biblioteca en el sistema de ficheros
DSVI libc.so	12.281.856 (11 MiB + 730 KiB)	12.293.171	12 KiB, 6 páginas	RW-, privada, tamaño fijo	Rellenar con ceros el marco de página asignado
mmap	12.294.144 (11 MiB + 742 KiB)	13.342.719	1 MiB, 512 páginas	RW-, privada, tamaño fijo	Fichero /home/pepe/datos.bin en el sistema de ficheros
Código libpthread.so	13.342.720 (12 MiB + 742 KiB)	13.435.349	92 KiB, 46 páginas	R-X, compartida, tamaño fijo	Biblioteca en el sistema de ficheros
DVI libpthread.so	13.436.928 (12 MiB + 834 KiB)	13.437.795	2 KiB, 1 página	RW-, privada, tamaño fijo	Biblioteca en el sistema de ficheros
DSVI libpthread.so	13.438.976 (12 MiB + 836 KiB)	13.447.327	10 KiB, 5 páginas	RW-, privada, tamaño fijo	Rellenar con ceros el marco de página asignado
...	...	...	...	...	...
Pila fun1	16.777.216 (16 MiB)	16.674.817	100 KiB, 50 páginas	RW-, privada, tamaño variable	Rellenar con ceros el marco de página asignado
Pila fun2	15.728.640 (15 MiB)	15.626.241	100 KiB, 50 páginas	RW-, privada, tamaño variable	Rellenar con ceros el marco de página asignado

Sistemas Operativos – 4o Semestre – GII & GMI

Cuarto Parcial. Comunicación y Sincronización. 1 de Junio de 2015

Un equipo de diseñadores software tiene como tarea el desarrollo de un ficticio sistema informático de la Junta electoral central para la noche de las elecciones. Este sistema está compuesto por múltiples actividades concurrentes, que deben coordinarse en el acceso a varias fuentes de información.

En cada colegio electoral se lleva a cabo el recuento de los votos. El equipo de diseñadores decide que éste sea el primer subsistema (Subsistema A).

Cada colegio electoral envía al coordinador de distrito la información sobre los votos utilizando acceso web seguro (*https*). Dicho coordinador se encarga de la integración de toda la información. El coordinador de distrito será el segundo subsistema (Subsistema B).

Finalmente, cada distrito envía a la Junta electoral central la información agregada y a partir de esta información, la Junta electoral lleva a cabo el cómputo global de los votos, almacenándolo en una Base de Datos. La Junta electoral central constituye el tercer subsistema (Subsistema C).

Centrándonos en el procesamiento del Subsistema A y teniendo en cuenta que el servicio se centraliza en una única máquina, el equipo de diseñadores baraja varias opciones:

A1. Diseñar una solución basada en un único proceso formado por diferentes threads, cada uno de los cuáles procesará una parte de los votos.

A2. Diseñar una solución basada en varios procesos pesados emparentados, cada uno de los cuáles procesará una parte de los votos.

A3. Diseñar una solución basada en varios procesos locales independientes, cada uno de los cuáles procesará una parte de los votos.

1. Para el diseño A1, indicar cuál de los siguientes mecanismos de comunicación es el más apropiado:
   a. Variables globales + mutex + condiciones
   b. Variables globales + semáforos
   c. Tuberías sin nombre (PIPE)
   d. Tuberías con nombre (FIFO)

Solución: Variables globales + mutex + condiciones

Dado que se trata de un único proceso formado por diferentes threads, éstos comparten el acceso a las variables globales, que deberá ser controlado mediante el uso de un mecanismo de sincronización adecuado. Los semáforos no son apropiados. Las mutex y condiciones se adaptan perfectamente a este escenario. Las tuberías no son necesarias.

2. Para el diseño A2, indicar cuál de los siguientes mecanismos de comunicación es el más apropiado:
   a. Variables globales + mutex + condiciones
   b. Memoria proyectada compartida sin semáforos
   c. Tuberías sin nombre (PIPE)
   d. Sockets de tipo datagrama con dominio AF_INET

Solución: Tuberías sin nombre (PIPE)

Dado que se trata de varios procesos pesados emparentados, las variables globales no son compartidas. Tampoco es factible utilizar memoria proyectada compartida sin utilizar ningún mecanismo de sincronización. Los sockets con dominio AF_INET permiten comunicar procesos en máquinas diferentes, por lo que no son requeridos. Por último, las tuberías sin nombre se adaptan perfectamente.

3. Para el diseño A3, cuál de los siguientes mecanismos de comunicación **no** es factible:
   a. Tuberías sin nombre (PIPE)
   b. Fichero proyectado en memoria + semáforos con nombre
   c. Fichero + cerrojos sobre fichero
   d. Sistema de Gestor de BBDD con transacciones

Solución: Tuberías sin nombre (PIPE)

Las tuberías sin nombre no pueden ser utilizadas para comunicar procesos independientes. El resto de mecanismos son factibles en el escenario descrito.

A continuación, el equipo se centra en el diseño de la comunicación entre el Subsistema A y el Subsistema B.

4. Si el número de colegios electorales por distrito no es muy elevado, ¿Qué arquitectura es la más adecuada para soportar dicha comunicación?
   a. Cliente/servidor con servidor con funcionamiento serie (un solo proceso atiende de forma secuencial a los clientes)
   b. Cliente/servidor con servidores dedicados (un proceso atiende a cada cliente)
   c. Arquitectura P2P (Peer-to-peer), con red estructurada
   d. Arquitectura P2P (Peer-to-peer), con red no estructurada

Solución: Cliente/servidor con servidores dedicados (un proceso atiende a cada cliente)

Dado que el número de colegios electorales no es muy elevado, el servidor de la solución Cliente/Servidor no sería un cuello de botella. Por tanto, la solución P2P la podemos descartar frente a C/S, pues complicaría demasiado el diseño de la solución. La solución C/S con funcionamiento serie causaría una contención innecesaria en los clientes. Por tanto, la solución C/S con servidores dedicados es la más adecuada.

5. ¿Cuál de los siguientes mecanismos de comunicación es adecuado para este escenario?
   a. Sockets de tipo stream con dominio AF_INET
   b. Sockets de tipo datagrama con dominio AF_INET
   c. Tuberías con nombre (FIFO)
   d. Sockets de tipo stream con dominio AF_UNIX

Solución: Sockets de tipo stream con dominio AF_INET

Las tuberías con nombre no permiten la comunicación entre procesos en diferentes máquinas. Tampoco lo permiten los sockets que utilizan dominio AF_UNIX. Por último, este escenario requiere fiabilidad, por lo que se requiere un socket de tipo stream con dominio AF_INET.

Para el desarrollo del Subsistema B, los diseñadores optan por una solución basada en procesos pesados. Uno de los diseñadores nos muestra la siguiente función utilizada en el subsistema B, que contiene errores:

```
void actualizar_votos (int votos, int id_colegio)
{

      struct flock fl;
      int fd;
      int val;
      int start = id_colegio*sizeof(int);

      fd = open("BD", O_RDWR | O_CREAT | O_TRUNC, 0644);
      fl.l_whence = SEEK_SET;
      fl.l_start = start;
      fl.l_len = sizeof(int);
      fl l_pid = getpid();
      fl.l_type = F_RDLCK;
      fcntl(fd, F_SETLK, &fl);
      lseek(fd, start, SEEK SET);
      read(fd, &val, sizeof(int));
      val += votos
      lseek(fd, start, SEEK SET);
      write(fd, &val, sizeof(int));
      fl.l_type = F_UNLCK;
      fcntl(fd, F SETLK, &fl);
      close(fd);
}
```

6. Indicar cuál de las siguientes acciones **no** es correcta de cara a resolver los problemas del código:
   a. El flag O_TRUNC debería eliminarse del código
   b. El cerrojo debería ser exclusivo
   c. La primera llamada a `fcntl` debería ser síncrona
   d. La segunda llamada a `lseek` debería eliminarse del código

Solución: La segunda llamada a lseek debería eliminarse del código

Si se elimina la segunda llamada a lseek, no actualizaremos el dato en la zona del fichero donde se guarda la información del colegio correspondiente, dado que la llamada read avanza el puntero de posición los bytes que corresponden al tamaño de un entero. Por otro lado, el flag O_TRUNC debería eliminarse, para evitar que el fichero se trunque cuando se lleva a cabo la apertura. El cerrojo tiene que ser exclusivo pues vamos a modificar la zona del fichero y la primera llamada a fcntl debe ser síncrona, para evitar que un proceso pueda progresar si no consigue el cerrojo exclusivo.

Respecto a la comunicación entre el subsistema B y el subsistema C, los diseñadores se plantean la siguiente cuestión:

7. Para que los subsistemas B y C puedan intercambiar mensajes de texto, ¿qué se requiere en ambos extremos de la comunicación?
   a. Convertir cada carácter del texto a enviar de formato host a formato de red en el subsistema B y de formato red a formato host en el subsistema C
   b. Convertir cada carácter del texto a enviar de formato de red a formato de host en el subsistema B y de formato de host a formato red en el subsistema C
   c. Transformar cada carácter del texto a enviar en coma flotante y utilizar un formato de conversión no estándar antes de que el subsistema B lo envíe al subsistema C.
   d. No llevar a cabo ninguna operación de conversión sobre los caracteres enviados.

Solución: No llevar a cabo ninguna operación de conversión sobre los caracteres enviados

Dado que la representación de un carácter no depende de la arquitectura de los subsistemas B y C, no es necesario llevar a cabo ninguna operación de conversión.

Los diseñadores deciden implementar el subsistema C como un servidor concurrente orientado a conexión, que espera las peticiones de los posibles clientes (procesos de diferentes subsistemas B). Uno de los diseñadores nos muestra el siguiente trozo de código del servidor, que contiene errores:

```
int sd, cd, size;
struct sockaddr_in s_ain, c_ain;
sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
bzero((char *)&s_ain, sizeof(s_ain));
s_ain.sin_family = AF_INET;
s_ain.sin_addr.s_addr = INADDR_ANY;
s_ain.sin_port = 150;
bind(sd, (struct sockaddr *)&s_ain, sizeof(s_ain));
listen(sd, 100);

....
```

8. Indicar cuál de las siguientes respuestas corresponde a todos los errores del código:
   a. El número de puerto debería convertirse de formato host a formato red.
   b. Al campo `s_ain.sin_addr.s_addr` no se le puede asignar el valor `INADDR_ANY` , ya que debería tener asignado una dirección IP.
   c. El número de puerto debería convertirse de formato red a formato host.
   d. El número de puerto debería convertirse de formato host a formato red y al campo `s_ain.sin_addr.s_addr` no se le puede asignar el valor `INADDR_ANY` , ya que debería tener asignado una dirección IP.

Solución: El número de puerto debería convertirse de formato host a formato red

El único error viene dado por la no conversión de formato host a formato red (y no a la inversa) del número de puerto, que es un número entero. Al campo s_ain.sin_addr.s_addr se le puede asignar el valor INADDR_ANY, cuando queremos que el servidor acepte conexiones de cualquier cliente.

Sistemas Operativos – 4º Semestre – GII & GMI

Examen Final. 16 de Junio de 2015

Cuestión 1 **[1 pto]** Desarrolle clara y suficientemente "enlace físico vs. enlace simbólico, diferencias y similitudes".

Cuestión 2 **[1 pto]** Capacidad de direccionamiento de un inodo, ¿de qué parámetros depende y cómo se calcula?

- Hoja 1

Cuestión 3 **[1 pto]** Describa qué es un cambio de contexto e indique las causas por las que se puede producir en un sistema multitarea.

El cambio de contexto se produce cuando se pasa de ejecutar el proceso A a ejecutar el proceso B. Por un lado, se debe salvaguardar el estado en el que se encontraba ejecutando el proceso A y recuperar el estado en el que se detuvo la ejecución del proceso B para que pueda reanudar su ejecución en el mismo punto donde fue detenida. La intervención del SO provoca dos cambios de modo en la ejecución del procesador, de usuario a privilegiado y viceversa.

El cambio de contexto se puede producir voluntariamente por el propio proceso en ejecución o involuntariamente. En el primer caso, podría ser la invocación de un servicio bloqueante del sistema. En el segundo caso, se puede producir porque haya consumido la rodaja de tiempo asignada por el planificador del sistema o porque haya algún otro proceso del sistema con mayor prioridad que genere un evento que provoque la interrupción del proceso en ejecución. Por ejemplo, porque algún periférico complete la operación de entrada/salida solicitada previamente o porque haya vencido algún temporizador y haya que dar paso a la ejecución del proceso B, más prioritario que el A.

Cuestión 4 **[1 pto]** Especifique qué servicios del sistema o funciones de biblioteca alteran la imagen de memoria de un proceso (porque creen nuevas regiones de memoria durante su ejecución) e indique en cada caso el efecto que ha producido. * Solo los que aumentan tamaño

`fork`: duplica todas las regiones
`exec`: cambia la imagen de memoria del proceso en ejecución
`mmap`: proyecta un fichero en memoria
`dlopen`: crea las regiones de código y datos de la biblioteca que se carga en memoria
`pthread_create`: crea un nuevo thread y la correspondiente región asociada a su pila
`malloc`: dependiendo de la implementación, cuando se invoca por primera vez en un proceso, puede crear la región de heap si se gestiona de manera independiente de la región de datos

Problema 2

a) **[1 pto]** Indicar el resultado de la ejecución del siguiente código que monta dinámicamente la biblioteca `libm.so` e invoca una función contenida en esta biblioteca.

```
int main(int argc, char **argv){
void *handle;
double (*cosine)(double); //cosine es un puntero a función
handle = dlopen("/lib/libm.so",RTLD_LAZY);
if (!handle) {
        fprintf (stderr,"%s\n",dlerror());
        exit(1);
}
printf("%f\n",(*cosine)(2.0)); //Se obtiene el coseno de 2.0
dlclose(handle);
return 0;
}
```

Montaje biblioteca

1 dlopen
2 dlsym (punto de enlace)
3 dlclose

*dl → dinamic library

Solución

El montaje dinámico de la biblioteca `libm.so` se realiza correctamente, pero en la invocación de la función `cosine` el programa abortará su ejecución porque se producirá un error de acceso a una posición de memoria inválida al no haberse especificado el punto de enlace de la función dentro de la biblioteca mediante la invocación de la función `dlsym`.

b) **[1 pto]** Analice la ejecución del siguiente código. El fichero "fichero.txt" contiene la secuencia cíclica de caracteres "0123456789012..." y ocupa 1024 bytes. El tamaño de las páginas es de 4 KiB. Suponga que el fork está implementado con la optimización Copy-On-Write.

1. Indique el contenido del fichero tras la ejecución del código.

2. Describa el contenido y las características (tamaño, niveles de protección) de la región de memoria correspondiente al fichero proyectado desde su creación hasta su eliminación.

```
int main(int argc, char **argv){
char *orig,*p,*q; int i, fd, estado; pid_t pid; struct stat atrib;
fd=open("fichero.txt",O_RDWR);
fstat(fd,&atrib);
orig=mmap(NULL,atrib.st_size,PROT_WRITE,MAP_PRIVATE,fd,0);
close(fd);
pid=fork();
if (pid != 0) {
        p=orig;
        for (i=0;i<511;i++) {
                sleep(10);
                *p=(char)((i%10)+48); // 48 es el código numérico del carácter 0
                p++;
        }
        wait(&estado);
} else {
        q=orig+1000;
        for (i=0;i<4000;i++) {
                *q=(char)((i%10)+48); // 48 es el código numérico del carácter 0
                q++;
        }
        return 0;
}
munmap(orig,atrib.st_size);
return 0;
}
```

Solución

1. Al realizarse la proyección con el flag `MAP_PRIVATE`, los cambios realizados en la región de memoria no se reflejan en el fichero, por lo que el fichero conservará la misma información que tenía al inicio de la ejecución.
2. El fichero ocupa 1 KiB, que es un tamaño menor que el tamaño de una página. Por lo tanto, al invocar el servicio `mmap` se crea una región de tamaño 4 KiB. Recuerde que el tamaño de las regiones en memoria debe ser múltiplo del tamaño de la página del sistema. Al invocarse el servicio `fork`, con la optimización COW no se duplican las páginas de las regiones del proceso original hasta que no se produzca una escritura en la página que modifique su contenido. La página se marca de sólo lectura y con bit de COW. También se debe incrementar el contador de página al compartirse entre los dos procesos.
   Como el padre queda bloqueado durante 10 segundos, lo más probable es que el hijo entre en ejecución antes que el padre, y en el momento en el que realiza la primera iteración del bucle, se produce una escritura en una posición de memoria de la región original, provocando un error de protección, la creación de una copia privada y la asignación de un marco de página distinto del asignado en el padre para la región correspondiente al proceso hijo. Se decrementa el contador de página de la página asignada al proceso padre y se desactiva el bit de COW pues el contador ha llegado a 1. La ejecución del hijo continua hasta que se alcance la posición 4096 (i=3096), momento en el que se sale del rango direccionable dentro de la región del proceso y se produce un error por violación de memoria que provoca la finalización del proceso hijo y la liberación de todos los marcos de página utilizados por este proceso.
   Con la terminación del hijo, la ejecución del padre se reanuda. Ejecuta el bucle realizando las escrituras en el marco de página página inicialmente asignado en el servicio `mmap`, recoge el estado de terminación del hijo y desproyecta el fichero con el servicio `munmap`, liberándose el marco de página asignado a esa región.

c) **[1 pto]** Un proceso ha creado dos threads TA y TB desde el thread principal.

El TA contiene las siguientes líneas:	El TB contiene las siguientes líneas:
`act.sa_handler=tratar_alarma;` `sigaction(SIGALRM,&act,NULL)`	`act.sa_handler=SIG_DFL;` `sigaction(SIGALRM,&act,NULL)`

El thread principal del proceso invoca el servicio `alarm(2)` antes de la creación de los threads TA y TB. Especifique qué ocurre en los siguientes casos:

1. La alarma vence antes de crear los threads TA y TB.
2. Se crea, ejecuta y termina el thread TA; se crea, ejecuta y termina el thread TB; vence la alarma.
3. Se crea, ejecuta y termina el thread TB; se crea, ejecuta y termina el thread TA; vence la alarma.

4. Se crea, ejecuta y termina el thread TA; vence la alarma; se crea, ejecuta y termina el thread TB.
5. Se crea, ejecuta y termina el thread TB; vence la alarma; se crea, ejecuta y termina el thread TA.

Solución

1. Al recibirse la señal SIGALRM sin haberse armado se aplica el tratamiento por defecto, que consiste en la finalización del proceso.
2. El thread TB restaura el tratamiento por defecto tras la recepción de la señal SIGALRM, por lo que el proceso terminaría.
3. El thread TA arma el tratamiento de la señal SIGALRM, por lo que tras el vencimiento del temporizador se ejecutaría la función tratar_alarma y luego se continuaría la ejecución del thread principal en el punto donde se interrumpió con la recepción de la señal.
4. El thread TA arma el tratamiento de la señal SIGALRM, por lo que tras el vencimiento del temporizador se ejecutaría la función tratar_alarma y luego se continuaría la ejecución del thread principal en el punto donde se interrumpió con la recepción de la señal y se crearía el thread TB.
5. El thread TB especifica el tratamiento por defecto tras la recepción de la señal SIGALRM, por lo que el proceso terminaría y el thread TA no llegaría a crearse nunca.

Problema 1

Un servidor Web, en su proceso principal:

1. Atiende por el puerto 80 TCP, el protocolo HTTP (*Hypertext Transfer Protocol*) que permite solicitar la descarga de documentos de todo tipo (extensiones html, jpg, mp3, etc.).
2. Arranca un proceso pesado para dar servicio dedicado a cada solicitud de cada cliente y no espera a que termine.
3. Para evitar *zombies*, maneja `SIGCHLD` para esperar por los hijos según vayan terminando.

Cada servidor dedicado:

4. Recibe la solicitud (sobre un buffer de 4KiB) de su cliente y la decodifica llamando a la función de biblioteca `int HTTP_request(char*buffer,int size);` que devuelve un error `(-1)` o un descriptor `(>=0)` asociado al documento que solicita descargar.
5. Envía al cliente el contenido del documento solicitado, a trozos de 4KiB máximo y termina.

Sobre la función `HTTP_request`:

6. Una solicitud puede también referirse a un programa Web, esto es, a un documento que se ha de ejecutar en el servidor (extensiones php, asp, jsp, etc.) y cuya salida es el resultado a enviar.
7. En este caso `HTTP_request` determina la ruta local al mandato intérprete del lenguaje y la ruta local al programa a interpretar y devuelve el descriptor devuelto por la llamada a:
   `int open_command(char*interprete,char*programa);`

La función `open_command`:

8. Devuelve el extremo de lectura de un pipe al otro lado del cual deberá estar conectada la salida estándar de un proceso hijo creado para que ejecute el mandato intérprete con el programa Web como único argumento.

Se pide que codifique en C y para Unix partes fundamentales del servicio descrito.

a) **[1.5 puntos]** Implemente la función `servidor_Web` con las funcionalidades 1 a 3.
b) **[0.5 puntos]** Implemente la función `servidor_dedicado` con las funcionalidades 4 y 5.
c) **[1 punto]** Implemente la función `open_command` con la funcionalidad 8.

Solución

```
/* httpd.c */
/*
 * Copyright 2015 datsi.fi.upm.es.
 * Universidad Politécnica de Madrid
 * Autor: Francisco Rosales
 * Versión: 1.0        Fecha: 17 Jun 2015
 */
#define MYNAME "httpd"

#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void SIGCHLD_Handler(int signo);
int servidor_Web(void);
int servidor_dedicado(int cd);
int HTTP_request(char*buffer,int size);
int open_command(char*interp,char*progrm);


int main(int argc, char * argv[])
{
        return servidor_Web();
}


int servidor_Web(void)
{
        int sd, cd, size, ret;
        struct sockaddr_in s_ain, c_ain;

 /*3*/   SIGCHLD_Handler(0);

 /*1*/   sd = socket(AF_INET, SOCK_STREAM,
                    IPPROTO_TCP);
        if (sd == -1) {
                perror(MYNAME);
                return 1;
        }

 /*1*/   bzero((char *) &s_ain, sizeof(s_ain));
 /*1*/   s_ain.sin_family = AF_INET;
 /*1*/   s_ain.sin_addr.s_addr = INADDR_ANY;
 /*1*/   s_ain.sin_port = htons(80);

 /*1*/   ret = bind(sd, (struct sockaddr*)
                    &s_ain,sizeof(s_ain));
        if (ret == -1) {
                perror(MYNAME);
                return 1;
        }

        listen(sd, 5);

        while(1) {
                size = sizeof(c_ain);
 /*2*/            cd = accept(sd,(struct sockaddr*)
                            &c_ain, &size);
 /*2*/            switch (fork()) {
                case -1:
                        perror(MYNAME);
                        return 1;
                case 0:
 /*2*/                    close(sd);
 /*2*/                    ret = servidor_dedicado(cd);
 /*2*/                    close(cd);
 /*2*/                    exit(ret);
                default:
 /*2*/                    close(cd);
                }
        }
        return 0;
}

void SIGCHLD_Handler(int signo)
{
/*3*/   signal(SIGCHLD, SIGCHLD_Handler);
/*3*/   if (signo == SIGCHLD)
/*3*/           waitpid(-1, NULL, WNOHANG);
}


#define SIZE 4096
int servidor_dedicado(int cd)
{
        char buffer[SIZE];
        int size;
        int fd;

/*4*/   size = recv(cd, buffer, SIZE, 0);
        if (size == -1) {
                perror(MYNAME);
                return 1;
        }

/*4*/   fd = HTTP_request(buffer, size);
        if (fd == -1) {
                perror(MYNAME);
                return 1;
        }

/*5*/   while ((size = read(fd,buffer,SIZE))  > 0)
/*5*/      if ((size = send(cd,buffer,size,0) < 0)
                break;
        close(fd);
        if (size < 0) {
                perror(MYNAME);
                return 1;
        }
/*5*/   return 0;
}


int HTTP_request(char*buffer,int size)
{
        char interp[SIZE];
        char progrm[SIZE];
        ...
/*7*/           return open_command(interp, progrm);
        ...
}


int open_command(char*interp, char*progrm)
{
        int ret;
        int pp[2];

/*8*/   ret = pipe(pp);
        if (ret == -1) {
                perror(MYNAME);
                return -1;
        }

/*8*/   switch (fork()) {
        case -1:
                close(pp[0]);
                close(pp[1]);
                perror(MYNAME);
                return -1;
        case 0:
/*8*/             dup2(pp[1],1);
/*8*/             close(pp[0]);
/*8*/             close(pp[1]);
/*8*/             execlp(interp,interp,progrm,NULL);
                perror(MYNAME);
                exit(1);
        default:
/*8*/             close(pp[1]);
        }
/*8*/   return pp[0];
}
```

Sistemas Operativos 4º Semestre. Grados II y MI

Primer Parcial. Sistema de Ficheros. 12 de Marzo de 2015.

Dispone de 60 minutos. Publicación: el 26 de Marzo. Revisión: el 10 de Abril.

Ejercicio único

Los siguientes apartados están relacionados.

A) (2 puntos) Implemente en C y para UNIX el mandato `incrementa` *`archivo`*, que:

A1) Abre el archivo indicado como argumento y redirige la entrada y la salida estándar a dicho archivo.

A2) Seguidamente, a través de los descriptores estándar, incrementa en 5 el entero binario (`int`) contenido en la posición 3 GiB del archivo.

B) (2 puntos) Según su implementación, ¿cómo se comportaría `incrementa` en los siguientes casos?:

B1) Si el archivo indicado fuese un fichero especial orientado a carácter, (ej. un terminal).

B2) Si el archivo indicado fuese un fichero normal con tamaño inicial 2 MiB. ¿Qué tamaño al final?

C) (2 puntos) Considerando que la cache de bloques del servidor de ficheros está inicialmente vacía y utilizando los mismos datos y bajo las mismas suposiciones del apartado **D**, conteste a las siguientes preguntas, donde nos referimos a las llamadas necesarias para implementar el mandato del apartado **A** (que incrementa el entero binario contenido en la posición 3 GiB de un archivo).

C1) Razone, ¿cuántos accesos a disco son necesarios para realizar la llamada `read` (del apartado **A**)?

C2) Razone, ¿cuántos accesos a disco son necesarios para realizar la llamada `write` (del apartado **A**)?

D) (2 puntos) Considere que el sistema de ficheros subyacente es de tipo UNIX (basado en inodos), con agrupaciones de 8 KiB, para un disco de medio TiB y archivos de tamaño medio igual a una agrupación. Justifique las suposiciones que considere necesarias.

D1) Exprese la capacidad de direccionamiento de un inodo en este sistema. Dé el resultado en bytes.

D2) Razone, ¿cuánto ocupará el mapa de bits de inodos en este sistema? Indique cálculos y unidades.

E) (2 puntos) Sea la siguiente visión parcial del contenido del sistema de ficheros:

NUM	T	U	G	O	USER	GROUP	PATH (RUTA)
1	d	rwx	r-x	r-x	root	root	/
2	d	rwx	rwx	rwt	root	root	/tmp/
3	-	r--	rw-	rwx	yoda	jedi	/tmp/datos.bin
4	d	rwx	r-x	r-x	root	root	/home/
5	d	rwx	r-x	---	r2d2	robot	/home/r2d2/
6	l	rwx	rwx	rwx	r2d2	robot	/home/r2d2/temporal -> /tmp/
7	-	rwx	r-s	---	r2d2	jedi	/home/r2d2/incrementa

Donde: NUM es el número de ruta; T es el tipo de objeto; U, G y O son los grupos de permisos (*user*, *group* y *other*); y la notación `A -> B` indica que el enlace simbólico `A` apunta a la ruta `B`.

Considere que el usuario `r2d2` (del grupo `robot`) invoca, desde su *home*, el siguiente mandato:

```
./incrementa   temporal/datos.bin
```

Para cada uno de los siguientes casos, conteste detallando cómo el sistema operativo decodificará la ruta indicada. Muestre: (número de) ruta, grupo de permisos y permiso concreto, que se validará a cada paso.

E1) ¿Podrá ponerse en ejecución el mandato indicado? ¿Qué identidad tendrá el proceso resultante?

E2) ¿Podrá el proceso abrir el archivo indicado? Detalle cada paso de la decodificación de la ruta.

Sea un proceso padre con un número indeterminado de hijos (al menos 1). Escribir un tramo de código fuente que permita al proceso padre esperar la terminación del proceso hijo con PID `p1` *y todos los que terminen antes. Si el hijo* `p1` *ha terminado involuntariamente por la llegada de la señal SIGKILL el proceso padre debe esperar la terminación del resto de hijos durante un tiempo máximo de 5 segundos y después terminar voluntariamente con código de terminación 2.*

NOTA: No usar ninguna variable `N`*, o similar, para indicar el número total de hijos.*

```
void f1(int s) { exit(2); }
int main (void) {
    int status;
    struct sigaction act;
    act.sa_handler = &f1;
    act.sa_flags = 0;
    sigaction (SIGALARM, &act, NULL);
    .....
    while (wait(&status) != p1)        → Devuelve el pid del proceso que ha muerto
        continue;
    if (WIFSIGNALED(status))           → ¿Ha muerto por una señal
      if (WTERMSIG(status) == SIGKILL){ → ¿Esa señal es SIGKILL?
        alarm(5);
        while (wait(&status) > 0)
            continue;
        exit(2)
      }
}
```

1. *Dado el siguiente código de un ejecutable cuyo nombre es* `run_processes`:

```
1  int main(int argc, char *argv[]){
2     int pid, number, status;
3     pid_t pid;
4     number = atoi(argv[1]);
5     if (number<=0)
6        return 0;
7     pid = fork();
8     if (pid == 0) {
9        fork();
10       number=number-1;
11       sprintf(argv[1], "%d", number);
12       execvp(argv[0],argv);
13       perror("exec");
14       exit(1);
15    } else {
16       while (wait(&status)!= pid)
17          continue;
18    }
19    return 0;
20 }
```

Aclaración: *La función* `atoi()` *devuelve el valor entero representado por la cadena de caracteres dada. Por otro lado, dado un valor entero y un buffer,* `sprintf()` *almacena en el buffer la tira de caracteres que representa el entero. Ejemplo:*

`atoi("35")` *devuelve 35.*

`sprintf(cadena, "%d", 35)` *almacena en la variable* `cadena` *el texto "35".* Opuesto al "atoi"

Se supone que los servicios fork() y execvp() se ejecutan de forma correcta.

a) *Indicar qué sucede si se ejecuta* `run_processes` *sin ningún argumento.* SIGEV Acceso incorrecto a memoria
   Para el resto del ejercicio considere la ejecución del mandato `run_processes 3`.
b) *Indicar qué hará el primer proceso hijo creado durante la ejecución de* `run_processes 3`.
c) *Dibujar el árbol de procesos resultantes de la ejecución de* `run_processes 3`.
d) *¿Cuál sería el número total de procesos creados durante la ejecución de* `run_processes 3`, *sin contar el proceso padre original?*
e) *Razonar si al realizar la ejecución de* `run_processes 3`, *podrá existir algún proceso huérfano y/o algún proceso zombie. Se considera proceso huérfano aquel cuyo padre ha muerto, y es adoptado por el proceso* `init`. (Si) (Si)
f) *Si se incluye delante de la línea 16 el siguiente código:*

```
     act.sa_handler = tratar_alarma;
act.sa_flags = 0;
sigaction(SIGALRM, &act, NULL);
alarm(10);
while (pid != wait(&status)) continue;
return 0;
```

Se declara dentro del main(), la variable: `struct sigaction act;` *y se declara como variable global* `int pid;`, *para que pueda ser vista por la función* `tratar_alarma`, *cuyo código se muestra a continuación:*

```
void tratar_alarma(void) {
   kill(pid, SIGKILL);
}
```

En el caso de que se lleve a cabo la ejecución `run_processes 3`, *¿cuál es el número máximo de procesos que podrían morir por la ejecución del servicio* `kill` *invocado en la función* `tratar_alarma`?

En los sitios donde hay un wait

1.14

El siguiente programa, denominado `watchdog_timer`*, nos permite comprobar cada cierto tiempo que un operario está alerta en su terminal de trabajo. Este programa recibe dos argumentos: (i)* `espera1`*, que nos indica cada cuánto tiempo hay que verificar la presencia del operario y (ii)* `espera2`*, que nos indica el tiempo máximo que tiene el operario para responder correctamente. El código del programa, a falta de completar algunas partes, es el siguiente.*

```
#define true 1
pid_t pid; // int pid;

void tratar_alarma()
{
   // Código de tratamiento de la alarma
}

int main (int argc, char **argv)
{
   struct sigaction act;
   int num, ch, espera1, espera2, numErrores;
   int valor;

   espera1 = atoi(argv[1]);
   espera2 = atoi(argv[2]);

   while (true) ——→ while(1){...}
   {
      sleep(espera1);

      pid = fork();
      switch (pid)
      {
         case -1:
            perror("Error en la llamada fork");
            exit(1);
         case 0:
            numErrores = 0;
            while (true)
            {
               num = generar_numero();
               printf("Introduzca el número %d", num);
               scanf("%d\n", &ch); ⟹ leer por el fdo
               if (ch==num)
                  return 0;
               numErrores++;
               if (numErrores==5)
                  exit(1);
            }
         default:
            act.sa_handler = tratar_alarma;
            act.sa_flags = 0;
            sigaction(SIGALRM, &act, NULL);
            alarm(espera2); ⟹ Se queda solo en el padre
            while (pid != wait(&valor));
               // Código para la comprobación de la finalización del proceso hijo
      } alarm(0);
   }
   return 0;
}
```

a) *¿Están manejando correctamente la alarma el proceso padre y los procesos hijo que se van creando?* Hay que desactivar la alarma alarm(0);

~~b) Indica el contenido correcto de la pila nada mas comenzar la ejecución de la tarea~~

c) *Para verificar la presencia del operario se crea un proceso hijo, y el proceso padre se queda bloqueado esperando por él. Sin embargo, es necesario verificar que el proceso hijo ha finalizado correctamente y no por la recepción de una señal. Codifique el fragmento de código que deberíamos añadir a nuestro programa después de la sentencia* `while(pid!=wait(&valor))`.

d) *En caso de que el operario no responda correctamente en el tiempo máximo establecido, se activa la función tratar_alarma, que debe matar al proceso hijo y, a continuación, ejecutar un programa denominado alarmon que no recibe ningún parámetro de entrada. Codifique la función tratar_alarma.*

e) *Queremos que nuestro programa no se vea afectado por las señales generadas desde teclado con* `Ctrl+C` *y con* `Ctrl+\`. *Codifique, utilizando una máscara de señales, una posible solución, teniendo en cuenta que tanto el proceso padre como los procesos hijo que se vayan creando deben estar protegidos ante dichas señales.*

f) *En esta ocasión, deseamos que tanto el proceso inicial como los procesos hijo estén protegidos ante las señales generadas con* `Ctrl+C` *y con* `Ctrl+\`, *pero no queremos proteger la ejecución del programa alarmon. Para resolver este problema, ¿sería posible codificar una solución basada en máscaras?*

c)

```
if (WIFEXITED(valor))
  if (WEXITEDSTATUS(valor) == 0)
    continue;
  else
    exit(1);
else if (WIFSIGNALED(valor))
    exit(2);
```

d)

```
void tratar_alarma(){
  kill(pid, SIGKILL);
  execlp("alarmon", "alarmon", NULL);
  exit(1)
}
```

e)

```
sigset_t mascara;
sigaddset(&mascara, SIGINT); ctrl+C
sigaddset(&mascara, SIGQUIT); ctrol+\
sigprocmask(SIG_SETMASK, &mascara, NULL);
```

las mascaras se heredan tanto en fork como en exec

No llega al proceso mientras este la máscara

Ejercicio 1

```
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7  #include <sys/wait.h>
8
9  pid_t hpid, ppid;
10 int i; int n; int estado;
11
12 int main (int argc, char **argv)
13 {
14
15  n = atoi (argv[1]);
16  ppid = getpid ();
17  fprintf (stdout, "Proceso padre: %d Proceso P:  %d\n", (int) getppid (),
18    (int) ppid);
19
20 for (i = 1; i < n; i++) {
21     hpid = fork ();
22     if (hpid == -1) {
23  perror ("P: fork"); exit (1);
24   }// if
25     if (hpid == 0) {
26  fprintf (stdout, "Hijo: %d pid: %d Padre: %d\n", i, (int) getpid (),
27   (int) getppid ());
28  }// if
29     else break;
30 }// for
31
32 if (hpid != 0) {
33     hpid = wait (&estado);
34     if (hpid == -1)
35  {
36  perror ("wait"); return 1;
37  }
38     else if (ppid != getpid ( )
39  {
40  fprintf (stdout, "Finalizado el hijo: %d\n", (int) getpid ());
41   return 0;
42  }
43     else fprintf (stdout, "Finalizado el proceso P: %d\n", (int) getpid ());
44  }
45  else if (i == n)
46    fprintf (stdout, "Finalizado el ultimo hijo: %d\n", (int) getpid ());
47
48  return 0;
49 }
```

pasa número el contenido de argv[1]

Da el pid del padre

Salgo del bucle for (Línea 32)

recibe el hpid del proceso al que espera cuando este acaba

Suponiendo que el proceso padre P tiene PID 100, que los procesos hijos incrementan en una unidad el PID del padre según se vayan creando y que durante la ejecución de P no se crean más procesos en el sistema:

a) *Obtenga el diagrama jerárquico de procesos creados indicando los PID de los procesos cuando se invoca la ejecución del proceso de la siguiente forma:* `./P 3`

b) *Indique una posible traza de ejecución obtenida con la invocación anterior.* ~~Zombie~~

c) *Si se elimina el servicio* `wait` *de la línea 33, ¿cómo afecta esta moficación a la terminación de los procesos?* Zombie

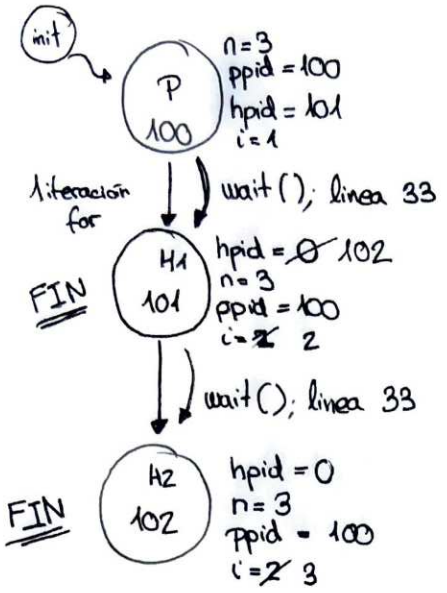
d) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para bloquear la recepción de la señal SIGINT en todos los procesos salvo en el proceso P, indicando dónde lo incluiría.* Se puede bloquear Línea 26

e) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para que los hijos ignoren la recepción de la señal SIGSTOP, indicando dónde lo incluiría.* No se puede bloquear

f) *Sobre la versión inicial del código adjuntado en el enunciado, añada el código necesario para que los hijos creen cada uno un thread de tipo ULT que ejecute el código asociado a la función* `void *terminar(void *p)` *en modo detached:*

```
void *terminar(void *p) {
 printf("Thread %u \n", (int)pthread_self());
 exit(0);
}
```

g) *Tomando como referencia el código desarrollado para el apartado anterior, indique el resultado de una posible traza de ejecución al invocar:* `./P 2`



Proceso Padre: xxx Proceso P: 100
Hijo1 pid: 101 Padre: 100
Hijo:2 pid: 102 Padre: 101
P H1 H2 línea 32

Finalizado último hijo 102
Finalizado hijo 101
Finalizado el proceso P: 100

Resumen - Sincronización y Comunicación

1. Semáforos

Servicios

***Inicialización y destrucción (Sin Nombre)**

```
int sem_init (sem_t *sem, int shared, int valini);
int sem_destroy(sem_t *sem);
```

***Inicialización y destrucción (Con Nombre)**

```
sem_t *sem_open (char *name, int flag, mode_t mode, int valini);
int sem_close (sem_t *sem);
int sem_unlink (char *name);
```

*** Decrementar e Incrementar**

```
int sem_wait (sem_t *sem);
int sem_post (sem_t *sem);
```

2. Mutex y Condiciones

Servicios

*** Inicializar y destruccion (NULL atributos por defecto) [Mutex]**

```
int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

*** Activar y desactivar [Mutex]**

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

*** Inicialización y destrucción [Condiciones]**

```
int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

***Liberacion, señal y broadcas [condiciones]**

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Explicación Condiciones

```
*Condition_wait (c,m) {
    mutex_unlock(m);
    //esperar aviso
    mutex_lock(m); }
```

```
*Condition_signal (c) {
    if (alguien espera)
        avisar que siga;
    else (se pierde); }
```

```
*Condition_broadcast(c) {
    while (alguien espera)
        avisa que siga; }
```

Explicación Mutex

```
mutex_lock(m) {
    if(no_hay_llaves)
        //esperar llave
    //Abrir cerrar y llevarmela
{
```

```
mutex_unlock(m) {
    if(alguien_esperando)
        //entregar llave
    else
        // Quedarmela
```

Ejemplo

*Main

```
int n_datos, buffer[1024];
pthread_mutex_t mutex;
pthread_cond_t no_lleno, no_vacio;        { Creamos los mutex y condiciones
int main (void) {
  pthread_t th1, th2;                     { Creamos los threads
  pthread_mutex_init (&mutex, NULL);
  pthread_cond_init (&no_lleno, NULL);    } Inicialización mutex y condiciones
  pthread_cond_init (&no_vacio, NULL);
  // Arranque de thread (create)
                        join
  pthread_mutex_destroy (&mutex);
  pthread_cond_destroy (&no_lleno);       } Destrucción mutex
  pthread_cond_destroy (&no_vacio);
  return 0;
}
```

3. Cerrojos (Protegen secciones de un fichero)

Servicio

* Inicialización y destrucción

```
struct flock {
   short l_type;   // F_RDLCK (compartido)
                   // F_WRLCK (exclusivo)
                   // F_UNLCK (elimina)
   short l_whence; // SEEK_SET, SEEK_CUR, SEEK_END
   off_t l_start;  // Desfase a l_whence
   off_t l_len;    // Tamaño 0
   pid_t l_pid;    // Solo F_GETLK, primer pid con lock
};

int fcntl (int fd, int cmd, struct flock *flockptr);
```

- cmd: F_GETLK ⇒ Comprueba si hay cerrojo
  F_SETLK ⇒ No Bloqueante, asíncrono
  F_SETLKW ⇒ Bloqueante, síncrono

Ejemplo

```
struct flock fl;
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 0; //Cierra todo el fichero
fl.l_pid = getpid();
```

```
fl.l_type = F_RDLCK; //Compartido
fcntl (fd, F_SETLKW, &fl);
lseek (fd, 0, sizeof(int)); //Nos colocamos al principio del principio
/*
-------------- */ Operaciones intermedias
fl.l_type = F_UNLCK;
fcntl (fd, F_SETLK, &fl);
```

4. Transacciones

* Idea conceptual de Cliente - Servidor (No puedes avanzar sin la respuesta de otro prog)

5. Mecanismos de Comunicación

- Mecanismo de Comunicación = Recurso Compartido + Mecanismo de Sincronización

Procesos/Hebras
- Procesos Pesados ⇒ Semáforos + Memoria Compartida
- Procesos Ligeros ⇒ Mutex y Condiciones + Imagen de memoria única

Procesos Pesados
- PIPEs y FIFOs
- Colas de mensajes
- Fichero + cualquier mecanismo de sincronización

Procesos/Remoto
- Sockets

6. Direccionamientos

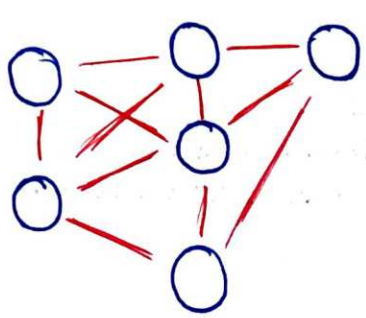
- Identifica
- Tipo: Sin Nombre (Id creador mecanismo)
  Nobre simbólico (www... .)
  Nombre físico (fd, IP TCP)
  Estructura de árbol
- Ámbito: Procesos Emparentados
  Local
  Remoto
- Servidor de nombres: texto → físico
  Local
  Remoto

Ejemplo: Cliente - Servidor

Cliente	Servidor
enviar(petición) →	while(1){
	recibir(petición)
	tratar();
recibir(petición) ←	enviar(petición)
	}

Petición →
← Respuesta

Ejemplo: Peer - to - Peer "P2P"



* Cada cliente funciona como servidor
- Escalabre
- Robusto
- Descentralizado
- Costes distrubuidos
- Anonimo
- Seguro

7. Mecanismos de Comunicación Local

PIPE

* Creación `int pipe (int fds[2]);`
* Lectura `int read (fds[0], datos, n);`
* Escritura `int write(fds[1], datos, n);`

FIFO

* Creación `int mkfifo (char *name, mode_t mode);`
* Eliminar `int unlink (char *name);`
* Abrir `int open (char *name, int flags);`
* Lectura `int read (fd_in, datos, n);`
* Escritura `int write (fd_out, datos, n);`

8. Mecanismo de Comunicación Remota Socket

Servicio principal

`int socket (int dominio, int tipo, int protocolo);`

* Dominio
  - AF_UNIX → Intra-maquina (DIR = nombre_fichero)
  - AF_INET → Entre_máquinas (DIR = dirección IP + nº de puerto)
  - Mismos servicios para todo dominio, pero diferente tipo de direcciones
* Tipo
  - Stream (SOCK_STREAM)
    - → Orientado a flujo de Datos
    - → CON conexión
    - → Fiable: asegura entrega y orden

    (Conversación Telefónica)
  - Datagrama (SOCK_DGRAM)
    - → Orientado a mensajes
    - → SIN conexion
    - → No Fiable: pérdida y desorden

    (Correspondencia Postal)
* Protocolo
  - Mensajes y reglas de intercambio
  - En AF_INET (Internet)
    - → IPPROTO_TCP: stream
    - → IPPROTO_UDP: datagrama

Caracteristicas

- Con nombre (dirección)
- Bidireccional
- Con buffering
- Bloqueante o no

Servicios

* Creación int socket (int dominio, int tipo, int protocolo);
* Socket → Local Dir int bind (int sd, struct sockaddr *dir, int tam);
* Socket → Remote Dir (cliente) int connect (int sd, struct sockaddr *dir, int tam);
* Preparar para aceptar conexiones (servidor) → int listen (int sd, int baklog);
* Aceptación de una conexión (servidor) → int accept (int sd, struct sockaddr *dir, int *tam);
* Transmisión → int send (int sd, char *mem, int tam, int flags);
  int recv (int sd, char *mem, int tam, int flags);

9. Interbloqueo

Un conjunto de procesos está en interbloqueo cuando cada proceso está bloqueado en espera de un recurso que está asignado a un proceso del conjunto

Resumen (I)

- La **concurrencia** sucede a muchos niveles
- Implica **compartición** de recursos
- **Condición de carrera** == No se puede garantizar la velocidad relativa de los procesos concurrentes ⇒ pueden suceder resultados incorrectos
- **Sección crítica** == trozos de código susceptibles de condición de carrera
- Para coordinar el acceso usamos:
  - Mecanismos de sincronización
  - Mecanismos de comunicación
- **Mecanismos de sincronización**:
  - Acciones **atómicas** sobre base de instrucciones máquina atómicas
  - Modelos clásicos: modelan arquitecturas clave en concurrencia

Para procesos fuertemente acoplados (== que comparten memoria):

- Semáforos == contador
- Mutex == cerrojo
- Condición: liberar temporalmente mutex para esperar indicación

Para procesos independientes

- Cerrojos sobre regiones de fichero

Transacciones: procesos independientes y distribuidos

Resumen (II)

- **Mecanismos de comunicación**
  == Recurso compartido + Mecanismo de sincronización
  - Con o sin nombre, descriptor de fichero o identificador propio, uni o bidireccional, con o sin *buffer*, bloqueante o no

Para procesos locales:

- PIPE == mecanismo para N productores y M consumidores
- FIFO == crear con `mkfifo`, usar como fichero, semántica de PIPE

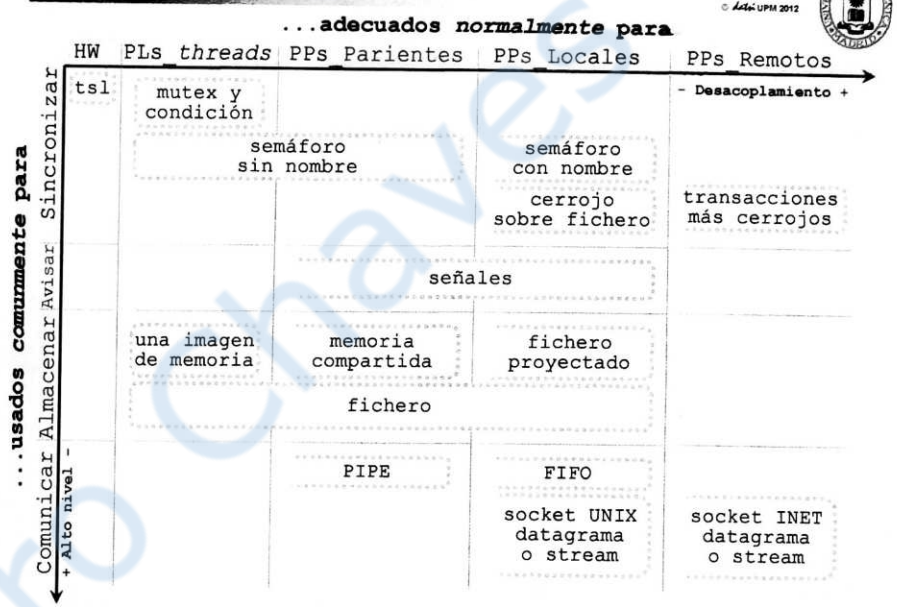
Para procesos remotos:

- *Sockets*: **dominio**, **tipo**, **protocolo**
  - Datagrama == correspondencia postal
  - *Stream* == conversación telefónica
- Servicios genéricos PERO direcciones específicas del dominio
- Formato de red

- **Interbloqueos**
  - Condiciones necesarias y tratamientos posibles

FIN

Mecanismos: uso y adecuación



EJEMPLO 1

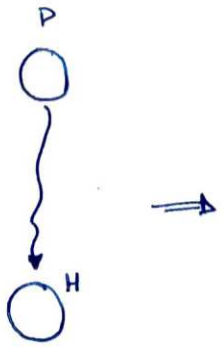
```
int n;
int a;

...
a = 5;

n = fork();
if (n == 0)
{
      a = a + 3;
      printf("%d\n",a);
}
else
{
      a = a + 2;
      printf("%d\n",a);
}

a = a + 7;
printf("%d\n",a);
```

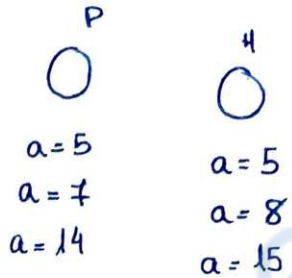
Hijo

Padre

Común

Hijo — Variables iguales





¿Qué valores imprime el Proceso Padre por pantalla? 7 → 14
¿Qué valores imprime el Proceso Hijo por pantalla? 8 → 15

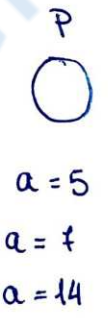
EJEMPLO 2

```
int n;
int a;

...
a = 5;

n = fork();
if (n == 0)
{
      execlp("programa", ...)
      a = a + 3;
      printf("%d\n",a);
}
else
{
      a = a + 2;
      printf("%d\n",a);
}

a = a + 7;
printf("%d\n",a);
```

Hijo

Padre

Común



exec no retorna !!

1 exec

Borra
- Imagen de memoria
- Descripción memoria
- Estado

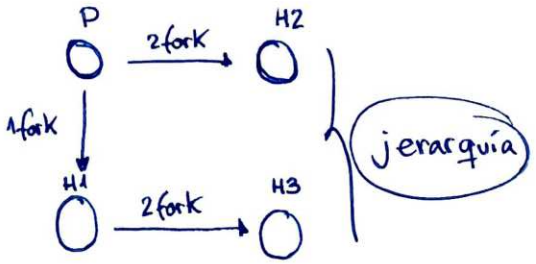
Conserva
- PID, PID padre...
- fd

carga datos del ejecutable

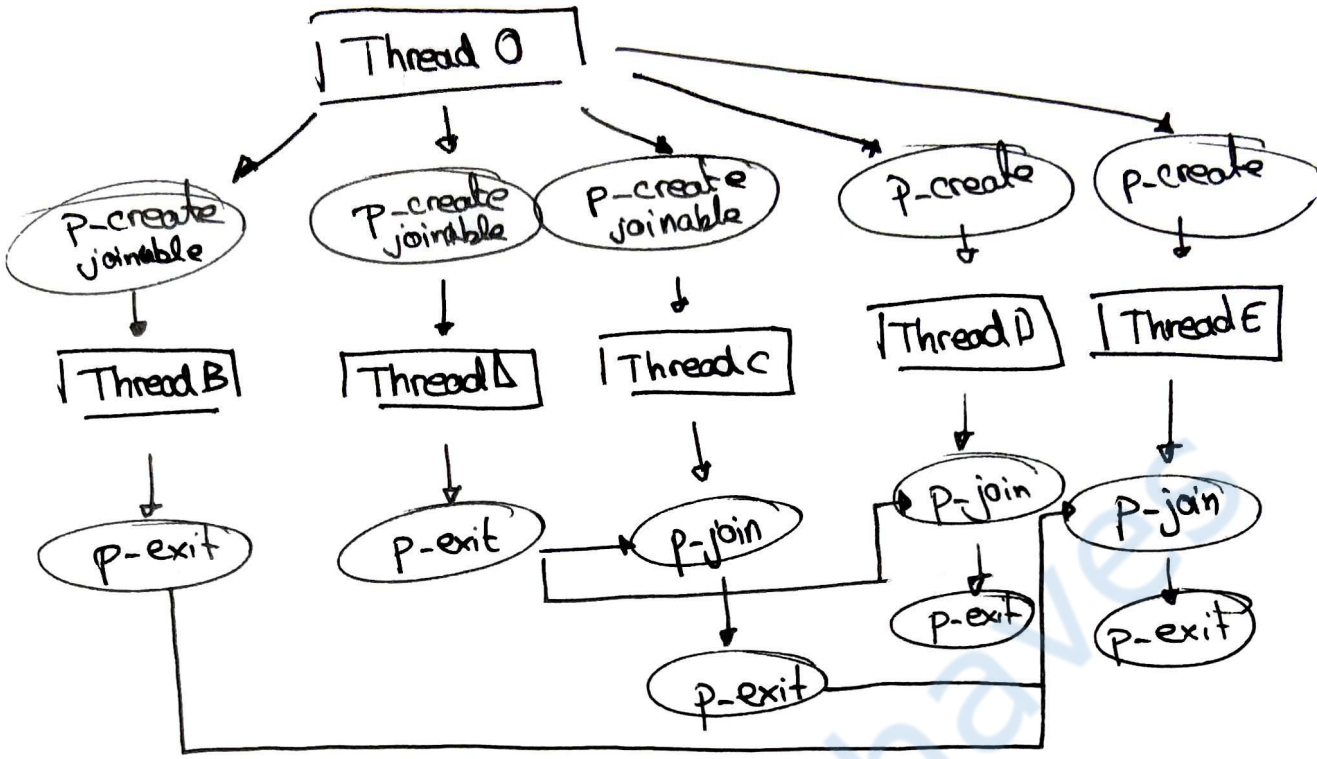
¿Qué valores imprime el Proceso Padre por pantalla? 7 → 14
¿Qué valores imprime el Proceso Hijo por pantalla? No imprime nada

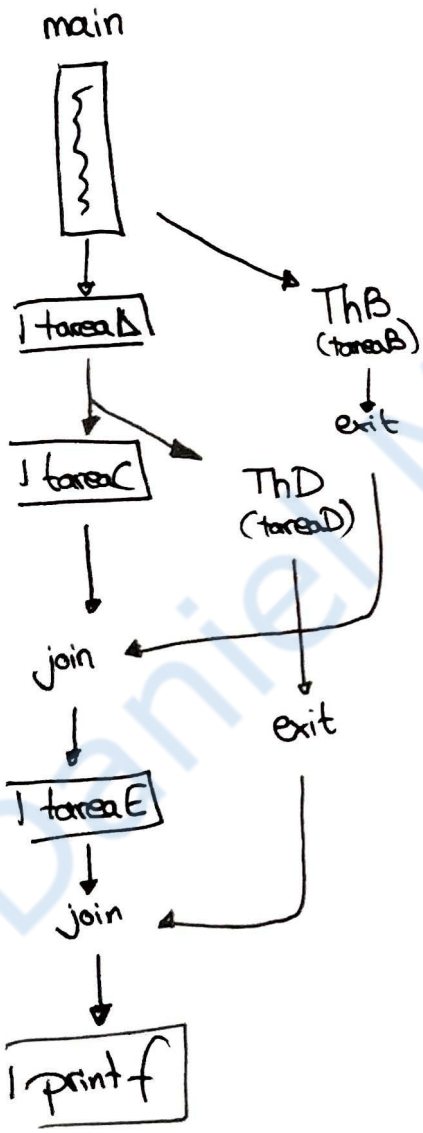
EJEMPLO 3
****

```
...
fork();
fork();
printf("hola\n");
...
```



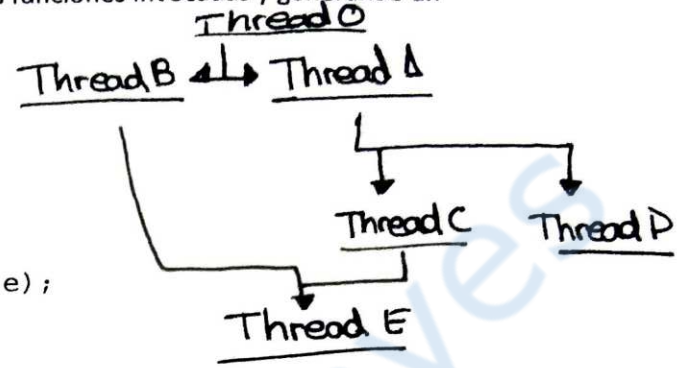
¿Cuántas veces se imprime el literal "hola" por pantalla? 4 : Según la planificación del SO cada proceso lo imprime el "hola" en un orden distinto
¿Qué jerarquía de procesos se obtiene?





Escribir un tramo de código que realice las mismas operaciones que el código que se muestra a continuación, pero maximizando la concurrencia entre las funciones invocadas y generando un único proceso.

```
int a, b, c, d, e;
a = tarea_A(0);
b = tarea_B(0);
d = tarea_D(a);
c = tarea_C(a);
e = tarea_E(b+c);
printf("%i %i %i %i %i\n", a, b, c, d, e);
```




```
#define MX_THREADS 5

int i;
pthread_attr_t attr;
pthread_t thid[MX_THREADS];
pthread_attr_init(&attr);
```

Sea un proceso padre con un número indeterminado de hijos (al menos 1). Escribir un tramo de código fuente que permita al proceso padre esperar la terminación del proceso hijo con PID p1 y todos los que terminen antes. Si el hijo p1 ha terminado involuntariamente por la llegada de la señal SIGKILL el proceso padre debe esperar la terminación del resto de hijos durante un tiempo máximo de 5 segundos y después terminar voluntariamente con código de terminación 2.

NOTA: No usar ninguna variable N, o similar, para indicar el número total de hijos.

Daniel Melero Chaves

Ejercicios de Procesos

EJERCICIO 2:

a) El pid del proceso hijo creado por la llamada fork()

b) El conjunto de sentencias que están englobadas en el bloque del "case 0:"

c) Comprueba si su pid es par o impar

d) Devuelve el valor "status" devuelto por el hijo a través del exit

e) Conjunto de sentencias de "default"

1) Incrementa el valor de K
2) Espera a que el hijo muera
3) Comprueba el estado de la muerte con WIFEXITED
4) Comprueba el status del status del hijo, si es 0 incrementa n

f) Solo 1 porque el padre no crea otro proceso hijo hasta que el que ha creado anteriormente muere

g)

1) K indica el nº de veces que se ha ejecutado el for
2) n indica el nº de veces que el pid del hijo es par

h) Sí, siempre que el pid del hijo creado sea impar

i) El nº de veces que se ha ejecutado el bucle for

j)

PID	N	K
		1
29180	1	2
29181	1	3
29182	2	4
29183	2	5
29184	3	

EJERCICIO 9:

a)

```
int main(void){
    void tarea_A(void);
    void tarea_B(void);
    return 0;
}
```

b)
```
int main(void){
    switch (fork()){
        case -1:
            perror("El hijo no ha sido creado bien");
            exit(1);
        case 0:
            tarea_A();
        default:
            tarea_B();
            wait(NULL);
    }
    return 0;
}
```

c)
```
int main(void){
    pthread_t thread;
    int pid;
    pthread_create(&thread, NULL, tarea_B, NULL);
    tarea_A();
    pthread_join(thread, NULL);
    return 0;
}
```

EJERCICIO 6

```
#include <curses.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int beep(void);

void tratar_alarma(){
    beep();
}
```

```
int main(int argc, char **argv) {

    int status;
    int tiempo;
    int i;
    sigset_t mascara;
    struct sigaction act;
    char car;
    char ultimoDigitoPid;
    pid_t pid;
    int resul;

    sigemptyset(&mascara);
    sigaddset(&mascara, SIGINT);
    sigaddset(&mascara, SIGQUIT);
    sigprocmask(SIG_SETMASK, &mascara, NULL);

    tiempo = atoi(argv[1])*60;

    while (1) {

        sleep(tiempo);
        pid = fork();
        switch(pid){

            case -1:
                perror("fork");
                exit(1);
            case 0:
                act.sa_handler = tratar_alarma;
                act.sa_flags = 0;
                sigaction(SIGALRM, &act, NULL);
                ultimoDigitoPid = '0' + getpid() % 10;
                printf("\nIntroduce %c", ultimoDigitoPid);
                for (i=0; i<15; i++){
                    alarm(2);
                    do{
                        resul = read(0, &car, 1);
                    }while (resul!=0 && car != ultimoDigitoPid);
                if(car == ultimoDigitoPid){
                    return 0;
                }
            }//for
            exit(1);
```

```
        default:
          wait(&status);
          if(status != 0){
            execvp(argv[1], &argv[2]);
            perror("exec");
            exit(1);
          }
      }
    }
    return 0;
}
```

EJERCICIO 7

```
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#define MAXBUF 512

pid_t pid1;
pid_t pid2;

void MatarProcesos(void){
    kill(pid1, SIGKILL);
    kill(pid2, SIGKILL);
    exit(0);
}

void Productor(int f){
    int fd, n;
    char buf[MAXBUF];
    fd = open("/tmp/datos.txt", O_RDONLY);
    if(fd == -1){ perror("open"); }
    else{
      while((n = read(fd, buf, MAXBUF)) != 0)
        write(f, buf, n);
```

```
        if (n == -1) { perror ("write"); }
    close (fd);
    return;
}
void Consumidor (int f) {
    int n;
    char buf [MAXBUF];
    while ((n = read (f, buf, MAXBUF)) != 0)
        write (1, buf, n);
    if (n == -1) { perror ("read"); }
    return;
}
int main (void) {
    int fd [2];
    struct sigaction act;
    if (pipe (fd) < 0) {
        perror ("pipe")
        return (0);
    }
    pid1 = fork ();
    switch (pid1) {
        case -1:
            perror ("fork 1")
            return 0;
        case 0:
            close (fd[0]);
            Productor (fd [1]);
            close (fd [1]);
            return 0;
        default:
            pid2 = fork ();
            switch (pid2) {
                case -1:
                    perror ("fork 2");
                    return 0;
```

```
            case 0:
                close(fd[1]);
                Consumidor(fd[0]);
                close(fd[0]);
                return 0;
            default:
                close(fd[0]);
                close(fd[1]);
                act.sa_handler = MatarProcesos;
                act.sa_flags = 0;
                sigaction(SIGALRM, &act, NULL);
                alarm(60);
                wait(NULL);
                wait(NULL);
        }
    }
    return 0;
}
```

1 EJERCICIO 5

```
int main(void){
pid_t p1, p2, p3;
int fd[2];
char buffer;
pipe(fd);
switch(p1 = fork()){
    case -1:
        perror("fork");
    case 0:
        close(fd[0]);
        write(fd[1], '1', 1);
        exit(0);
}
read(fd[0], &buffer, 1);
printf("p1 =", buffer);
```

```
        switch (p2 = fork()) {
            case -1:
              perror ("fork()");
            case 0:
              close (fd[0]);
              write (fd[1], '2', 1);
              exit 0;
        }
        read (fd[0], &buffer, 1);
        printf ("p2 =", buffer);
        switch (p3 = fork()) {
            case -1:
              perror ("fork()");
            case 0:
              close (fd[0]);
              write (fd[1], '3', 1);
              exit 0;
        }
        read (fd[0], &buffa, 1);
        printf ("p3 =", buffer);
        close (fd[1]);
        close (fd[0]);
        return 0;
}
```

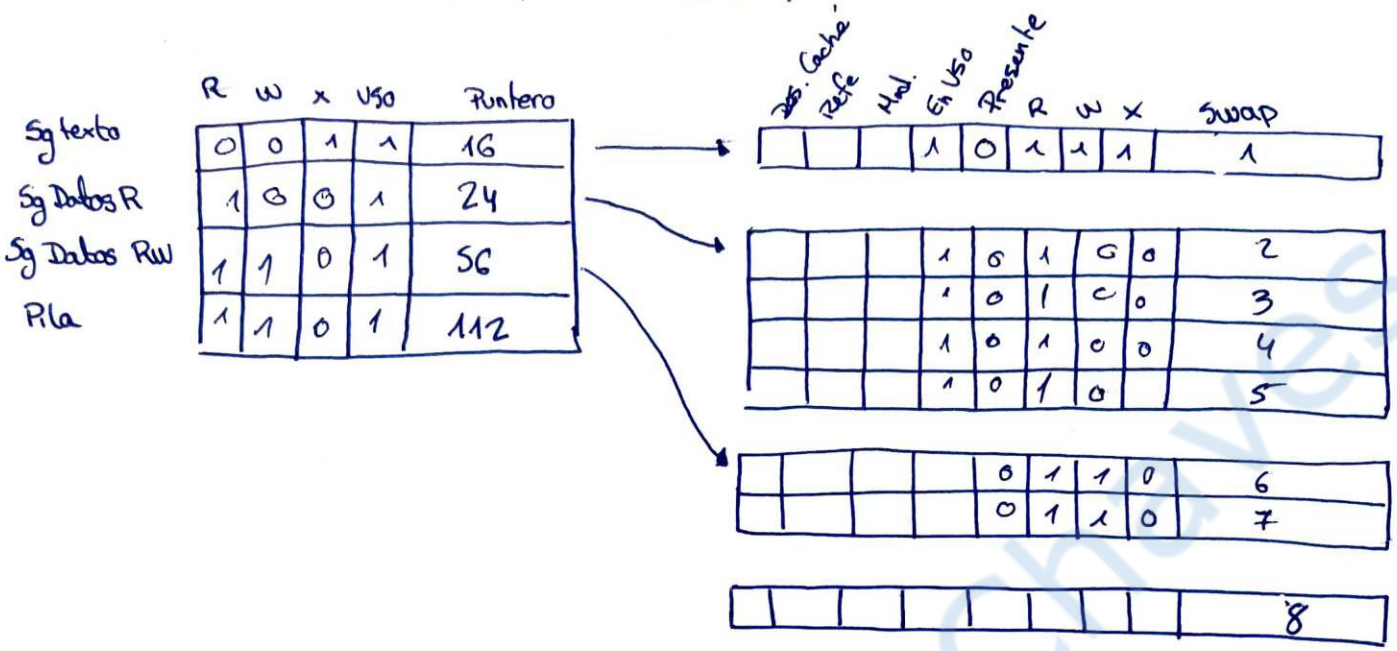
EJERCICIO 3

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {

    int i, fd[2];
    Char buffer;
```

```
for (i=0; i<n; i++){
    if(pipe(fd) <0) {
        perror ("pipe");
        exit(0);
    }
    switch (fork()) {
        case -1:
            perror ("fork()");
            exit(2);
        case 0:
            dup2(fd[0],0);
            close(fd[1]);
        default:
            dup2 (1, fd[1]);
            close(fd[0]);
            exit(0);
    }
}
return 0;
}
```

PROBLEMA 3.1

a) Para evitar huecos, vamos a emplear tablas multinivel



b) Al inicializar la ejecución no hay ninguna página. De hecho, la tabla de páginas propuestas en una sección anterior refleja esta situación. Dependiendo del SO, la pila inicial podría estar en un marco de página, puesto que es allí donde la crea el SO.

Al intentar ejecutar el proceso dará un fallo de página que hará traer a un marco de memoria principal la página del texto. Seguidamente, se irán produciendo otros fallos de página que harán que otras páginas deban pasar a marco.

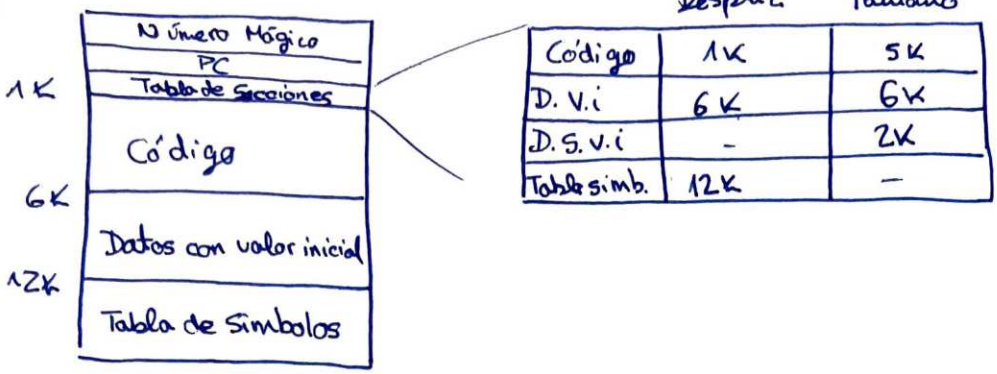
c) El proceso intenta ejecutar PUSH. Según crezca la pila se incrementa o decrementa el SP, lo que genera una dirección virtual que corresponde a una página inexistente. La MMU recibe esta dirección para traducirla y, al acceder a la tabla de páginas, detecta que se sale del rango. La MMU genera un trap de vidación de memoria.

El SO trata el trap y para ello tiene 2 posibilidades:

- Si existe espacio disponible se asigna más memoria al proceso
- Si no existe espacio disponible se para la ejecución del proceso.

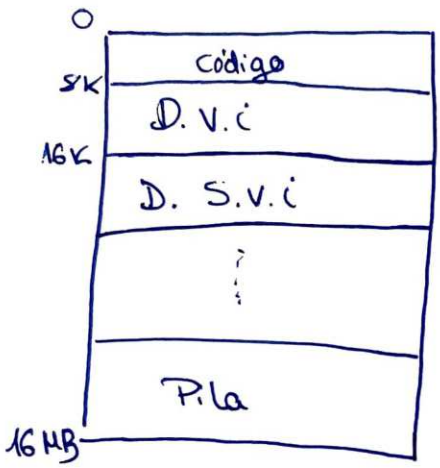
Problema 3.2

a)



	Desplaz.	Tamaño
Código	1K	5K
D. V.i	6K	6K
D. S. V.i	-	2K
Tabla simb.	12K	—

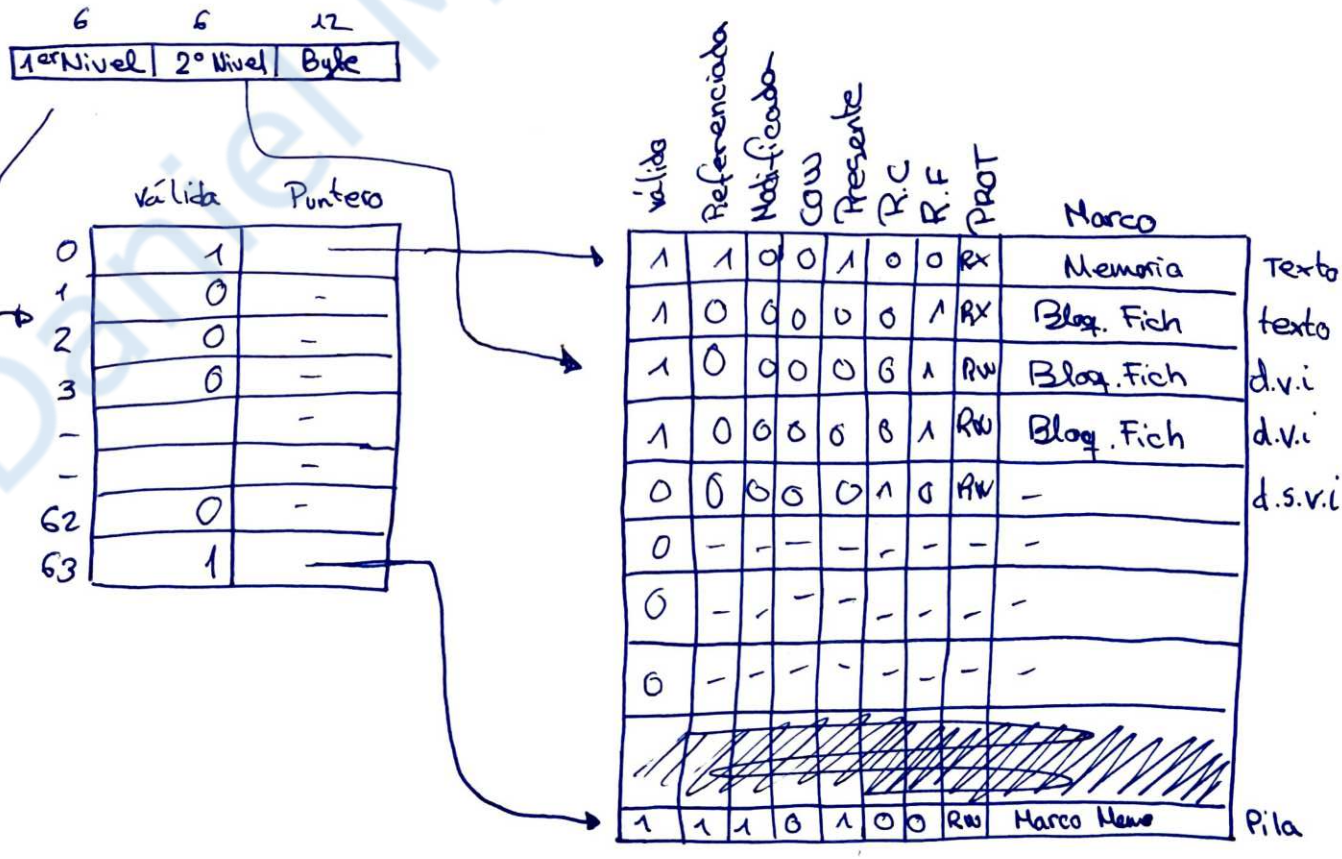
b)



c) Debido a que tenemos un esquema de planificación, se produce fragmentación interna. Puede desperdiciarse parte de último marco asignado cada región del proceso

Problema 3.3

a)



b) Espacio de Swap: la 1ª vez se le reserva el espacio en el swap, las siguientes veces usará el espacio que se le reservó la primera vez. El bit de modificado lo inicializa el SO cada vez que llega una página de memoria principal. Es escrito por la MMU cuando hace un acceso de escritura a la página. Sirve para saber si la página está sucia, a la hora de expulsarla a memoria secundaria.

Problema 3.4

a) se aplican los siguientes cambios tomando como referencia el dibujo de Problema 3.3 apartado a)

Válida 1, Puntero 1 se añade la siguiente tabla

④ 1 1 1 0 1 0 0 RW Marco Memoria (d.s.v.i)

y en la tabla del 2º nivel

⓪ 1 1 0 0 1 0 0 RX Marco Memoria - (Cod. Bid)
① 1 0 0 0 0 0 1 RX Bloq Fich (Cod Bid)
② 1 1 1 0 1 0 0 RW Marco Mem (Datos Bid)
③ 1 0 0 0 0 0 1 RW Bloq Fich (Datos Bid)
④ 1 0 0 0 0 0 1 RW Bloq Fich (Datos Bid)

b) Inicialmente la región se comparte con COW y el hijo, que escribe primero, crea una copia privada cuando va a modificar los datos.

Problema 8.12

① FIFO y 3 marcos de página

PET	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M1	4	4	4	5	5	5	3	3	3	3	3	3	3	3	3
M2	-	2	2	2	4	4	4	4	4	4	4	4	1	1	1
M3	-	-	1	1	1	2	2	2	2	2	2	2	2	5	5
FP	P	P	P	P	P	P	P	-	-	-	-	-	P	P	-

② FIFO y 4 marcos de página

PET	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M1	4	4	4	4	4	4	3	3	3	3	3	3	3	5	5
M2	-	2	2	2	2	2	2	4	4	4	4	4	4	4	3
M3	-	-	1	1	1	1	1	2	2	2	2	2	2	2	2
M4	-	-	-	5	5	5	5	5	5	5	5	5	1	1	1
PP	P	P	P	P			P	P	P				P	P	P

③ LRU y 3 marcos de página

M1	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M2	4	4	4	5	5	5	3	3	3	3	3	3	1	1	1
M3	-	2	2	2	4	4	4	4	4	4	4	4	4	5	5
M4			1	1	1	2	2	2	2	2	2	2	2	2	3
PP	P	P	P	P	P	P	P						P	P	P

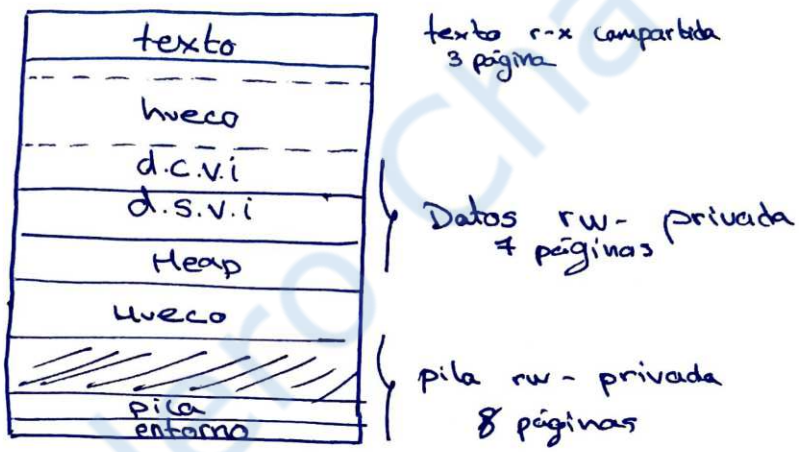
④ LRU y 4 marcos de página

M1	4	2	1	5	4	2	3	4	2	3	4	2	1	5	3
M2	4	4	4	4	4	4	4	4	4	4	4	4	4	4	3
M3	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2
M4	-	-	1	1	1	1	3	3	3	3	3	3	3	5	5
M5	-	-	-	5	5	5	5	5	5	5	5	5	1	1	1
PP	P	P	P	P			P						P	P	P

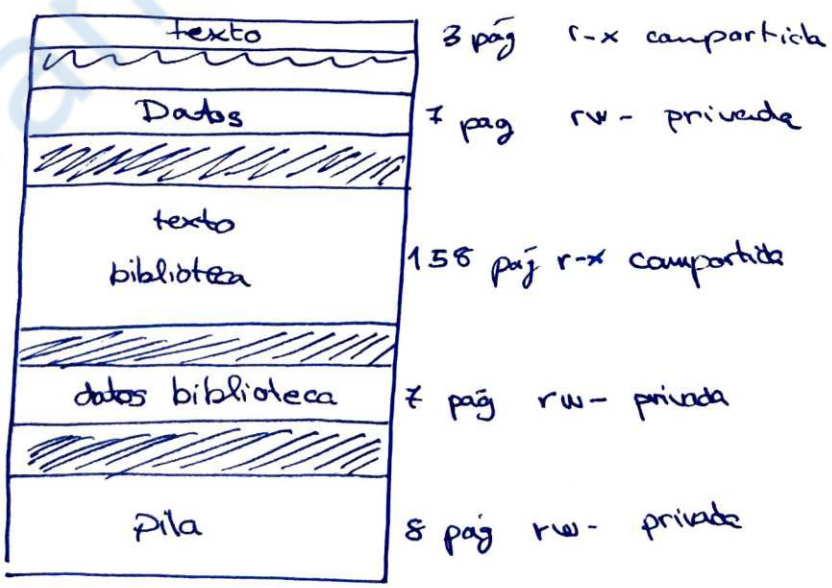
Problema 3.8

a) Se necesitan 3 páginas puesto que 3×4 KiB = 12.288 B > 12.096.
Esta región es compartida y de tamaño fijo, tiene permisos de lectura y ejecución, y su fuente es ejecutable

- Campo datos = 1.080 B + 8.724 B + 16 KiB = 9.804 B + 16 KiB, por tanto se requieren 7 páginas
- La pila tiene un tamaño de 32 KiB



b) La biblioteca requiere 2 regiones, una de texto y otra de datos.
Los datos de la biblioteca son 645.024 B por lo que necesitamos 158 páginas = 647.168 B.
Para los datos, que ocupan 28.144 B, necesitaremos 7 páginas = 28.672 B



Problema 3.9

a) El padre y el hijo escriben en la región compartida, por lo que los valores de dicha región dependen del orden de ejecución.
Se pueden dar los siguientes valores:

11 - 33 - 13 - 31

↳ Los más probables

b)

Al final del 1º bucle = 01....

Al final del 2º bucle = 0156789....

c)

```
void *h;
int (*max) (int, int);
h = dlopen ("/lib/mis_numeros.so", RTLD_LAZY);
max = dlsym (h, "mi_maximo");
printf ("%d\n", (*max)(3,5));
```

Melero Chaves, Daniel

Comunicación y Sincronización

Ejercicio 1

(a) Se trata de un modelo concurrente, porque atiende varias peticiones de clientes. Para poder realizar esto de manera organizada, cada cliente debe esperar su turno para ser atendido.

(b)

Cliente

```
int main (int argc, char *argv[]){
    int cliente_id;
    struct sockaddr_in dir_cliente; //Estructura de AF_INET
    unsigned char byte; //Guardamos el Valor
    bzero ((char *) &dir_cliente, sizeof(dir_cliente)); //Escribe todo 0s en dir_cliente
    dir_cliente.sin_family = AF_INET;
    dir_cliente.sin_port = htons(7); //número de puerto [enteros de formato host a red]
    if (cliente_id = socket (AF_INET, SOCK_STREAM, IPPROT_TCP) <0) //Creación del socket
        fprintf (stdout, "ERROR EN EL SOCKET");
    if( connect (cliente_id, (struct sockaddr *) & dir_cliente, sizeof (dir_cliente)) <0)
        fprintf (stdout, "ERROR EN LA CONEXION");
                                        //Asocia a una dir remota
    while (read (0, &byte, 1) ==1){
        send (cliente_id, &byte, 1, 0);
        recv (cliente_id, &byte, 1, 0);
        write(1, &byte, 1); }
    close (cliente_id); //Cerramos el socket
}
```

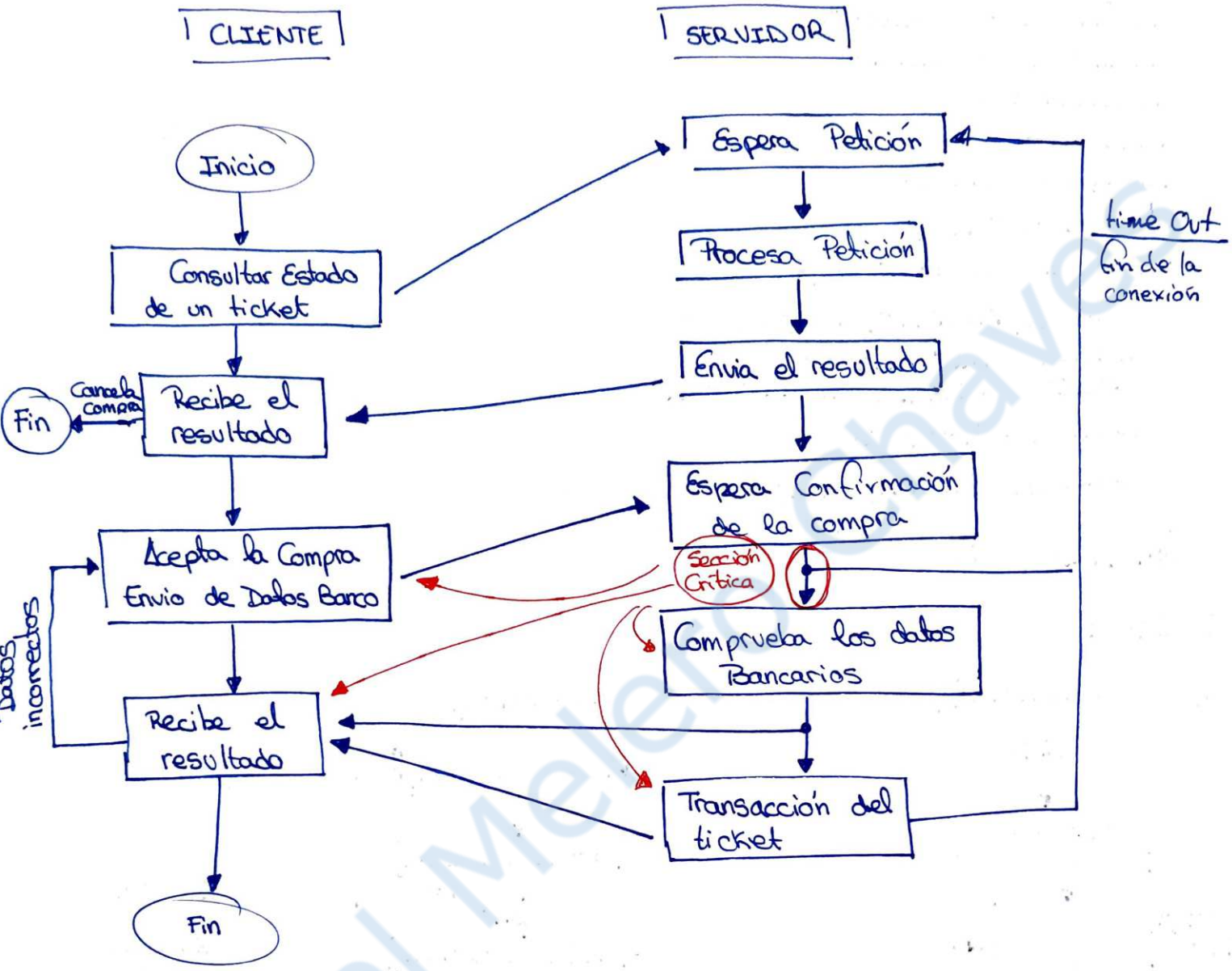
Proceso de diálogo con el servidor, nos comunicamos con lo asignado al servicio socket

Servidor

```
int main (void){
   int server_id, cuenta_id, tamano;
   unsigned char byte;
   struct sockaddr_in server_dir, cliente_dir; // Estructura de AF_INET { Servidor / Cliente
   if (server_id = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP) < 0) // Socket de comunicación del server
      fprintf (stdout, "ERROR EN EL SOCKET");
   bzero ((char *) & server_dir, sizeof (server_dir)); //Ponemos todo a 0s server_dir
   server_dir.sin_family = AF_INET;
   server_dir.sin_port = htons (7);
   server_dir.sin_addr.s_addr = INADDR_ANY; // Aceptar cualquier dirección
   if (bind (server_id, (struct sockaddr *) & server_dir, sizeof (server_id)) < 0)
      fprintf ( stdout, "ERROR EN EL BIND");   //Asociar a una dir local
   listen (server_id, 100) /* 100 tamaño de la cola del servidor */
   while (true) {
     tamano = sizeof (cliente_dir);
     cliente_id = accept (server_id, (struct sockaddr *) &cliente_dir, sizeof (cliente_dir));
     switch (fork()) {   //Aceptar una conexión
          case -1:
            perror ("ERROR EN EL FORK");
            return 1;
          case 0 : // Servidor hijo atiende la petición del cliente, acaba, muere
            close (server_id);
            while (recv (cliente_id, &byte, 1, 0) == 1)
                 send (cliente_id, &byte, 1, 0);
            close (cliente_id);
            return 0;
          default: //Servidor padre deja de atender la petición
            close (cliente_id);
     } /* switch*/
   } /* while*/
} /* Main */
```

Ejercicio 2

(a)



(b) Para realizar un registro necesitamos:

- Comprobar información del cliente
- Fecha de compra
- Identificador del ticket
- Datos de la máquina

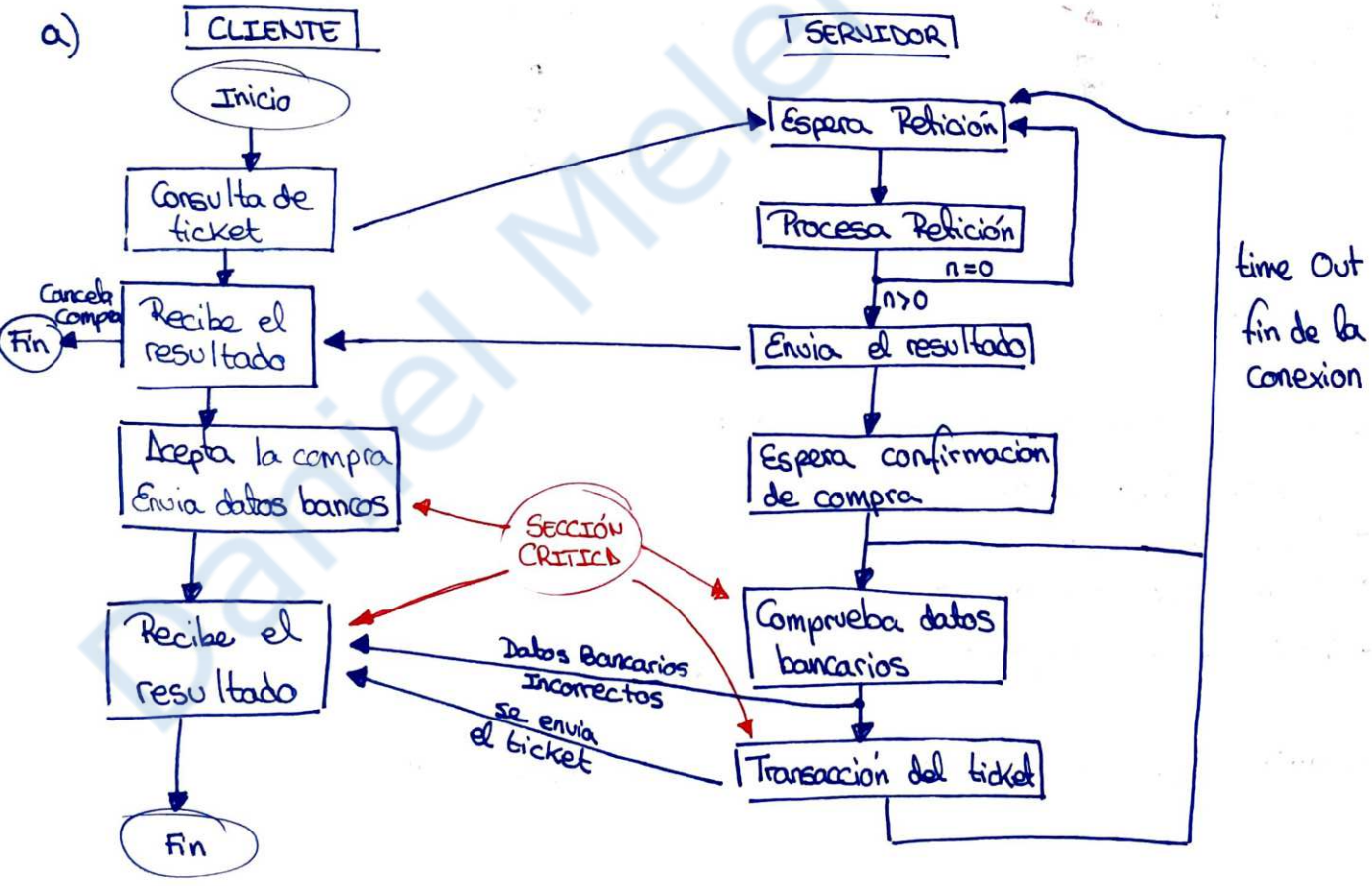
(c) Cerrojos (Procesos Pesados y estructura Cliente-Servidor)

Ejercicio 3

```
Empezar_transaccion
int fd;
struct flock fl;
fd = "Abro datos bancarios";
fl.l_whence = SEEK_SET;
fl.l_start = 0;
fl.l_len = 0;
fl.l_pid = getpid();
fl.l_type = F_WRLCK;
if (ticket.estado == LIBRE){
   fcntl(fd, F_SETLKW, &fl); //Pongo el cerrojo
   REALIZO TRANSACCIÓN;
   CAMBIO  TICKET.estado == OCUPADO; //Cambio el ticket de libre a ocupado
   fl.l_type = F_UNLOCK;
   fcntl(fd, F_SETLK, &fl); //Quito el cerrojo
}
END_TRANSACCIÓN
```

Ejercicio 4

a)



b) Para realizar un registro necesitamos
- Comprobar información del cliente
- Fecha de compra
- Nº del ticket
- Datos de la máquina

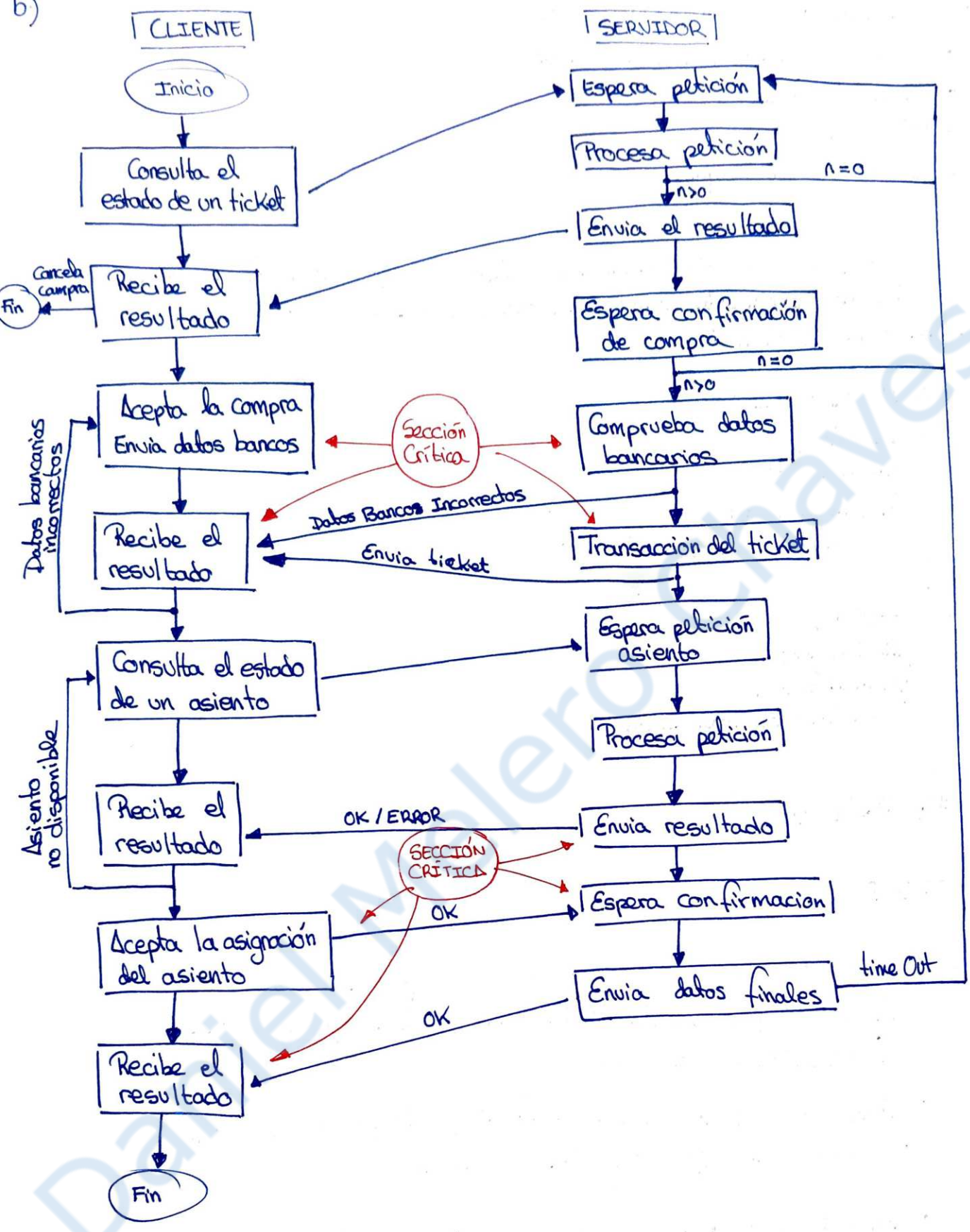
c) Implementaremos el código mediante semáforo, ya que como éstos tienen memoria, nos permitirán controlar la variable del número de veces que puede venderse un ticket

d)
```
BEGIN TRANSACION
  sem_wait {
    ntickets --;
    if (ntickets == 0)
      cancelamos transacción
  }
  sem_post {
    ntickets ++;
    if(ntickets > 0)
      realizamos transacción
  {
END TRANSACION
```

Ejercicio 5

a)
- El ejercicio presenta como inconveniente que entre la compra del ticket y la selección del asiento tiene que haber una actualización de los datos.
- Todos los asientos tienen el mismo precio porque la compra se hace antes que elección del asiento
- Con este nuevo planteamiento del programa hay 2 secciones críticas, la anterior y la elección del asiento.

b)



c) Para realizar un registro necesitamos

- Comprobamos información del cliente
- Fecha de compra
- Identificador del ticket
- Datos de la máquina
- Identificador del asiento